NUMERIC PROGRAM ANALYSIS TECHNIQUES WITH APPLICATIONS TO ARRAY ANALYSIS AND LIBRARY SUMMARIZATION

by

Denis Gopan

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2007

To my daughter, Yunna.

ACKNOWLEDGMENTS

First of all, I would like to thank my adviser Thomas Reps. Under his guidance, I learned a great deal about program analysis and software verification, as well as about other related areas of computer science. More importantly, Tom has taught me how to address problems and how to express my thoughts in writing in a way comprehensible to others. I only hope that at least a small fraction of Tom's dedication to research had rubbed off on me.

I would like to thank Ras Bodik, who was my academic adviser early in my graduate-student career, and Mooly Sagiv, with whom I had a pleasure of collaborating on several projects. Both Ras and Mooly played important roles in my development as a scientist. Also, I am thankful to Bertrand Jeannet, who taught me to appreciate the more formal (and the more arcane) aspects of abstract interpretation; to Ethan Munson, who encouraged me to apply to graduate school; and to Michael Gontar, who was both my companion and my guide when I took the first steps into the area of Computer Science.

I would like to thank the members of my Ph.D. committee, Somesh Jha, Ben Liblit, Marvin Solomon, and Amos Ron, for their comments on my thesis and for the insightful questions they raised during my defense.

During my graduate studies, I was fortunate to have a number of amazing officemates who had a profound influence on me both inside and outside of my research. I am thankful to each and every one of them: Shai Rubin, Glen Ammons, Darryl Roy, Michael Brim, Nick Kidd, Alexey Loginov, and Evan Driscoll. Also, I would like to thank the members of programming languages research group and security research group at Wisconsin: Susan Horwitz, Charles Fischer, Somesh Jha, Ben Liblit, Glen Ammons, Gogul Balakrishnan, Mihai Christodorescu, Vinod Ganapathy, Nick Kidd, Raghvan Komondoor, Akash Lal, Junghee Lim, Alexey Loginov, David Melski, Anne Mulhern, Manoj Plakal, Shai Rubin, Cindy Rubio, Hao Wang, Suanhsi Yong, and others. I am thankful to Mooly's students Tal Lev-Ami, Greta Yorsh, and others in Tel Aviv for very interesting (but, unfortunately, very rare) discussions.

I would like to thank my daughter Yunna and my wife Julia for their unconditional love and moral support. They provided the inspiration for me to complete this dissertation.

I am indebted to my parents Tatyana and Alex for their constant support and their belief in my abilities. If it was not for them, I would have never made it this far. I am especially grateful to my grandfather Anatoly whose dream was to see me become a researcher. I wish he had lived long enough to see the day. I also would like to thank my sisters Olga and Ellen, my aunt Svetlana, my grandma Lisa, my cousin Vitaliy, my uncle and aunt Lev and Nataliya, and other members of my family: they all played an important role in shaping my personality.

Last, but not least, I would like to thank my friends, Shura and Nata, Rost, Sasho, Taras and Tamara, Alex and Maya, Vadim and Anya, Lev and Erica, Dmitri and Ira, Sergei and Julia, Alexey and Wendy, Liya, Igor Solunsky, Zurab and Julia, Alexey Samsonov, Rita and Seich, and many, many others, for their support, encouragement, and for providing welcome distractions from my research work.

I am sure that I am forgetting to mention some people who have contributed in some way to the completion of my dissertation. To those people, please know that I am grateful.

Funding. My dissertation research was supported by a number of sources, including the Office of Naval Research, under grant N00014-01-1-0796, the UW Graduate School, under a Cisco Systems Distinguished Graduate Fellowship, and the National Science Foundation, under grant CCF-0540955. I am grateful for their support.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Appendix

Page

Appendix

Page

DISCARD THIS PAGE

LIST OF TABLES

Table Page

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

ABSTRACT

Numeric program analysis is of great importance for the areas of software engineering, software verification, and security: to identify many program errors, such as out-of-bounds array accesses and integer overflows, which constitute the lion's share of security vulnerabilities reported by CERT, an analyzer needs to establish numeric properties of program variables. Many important program analyses, such as low-level code analysis, memory-cleanness analysis, and shape analysis, rely in some ways on numeric-program-analysis techniques. However, existing numeric abstractions are complex (numeric abstract domains are typically non-distributive, and form infinite-height lattices); thus, obtaining precise numeric-analysis results is by no means a trivial undertaking.

In this thesis, we develop a suite of techniques with the common goal of improving the precision and applicability of numeric program analysis. The techniques address various aspects of numeric analysis, such as handling dynamically-allocated memory, dealing with programs that manipulate arrays, improving the precision of extrapolation (widening), and performing interprocedural analysis. The techniques use existing numeric abstractions as building blocks. The communication with existing abstractions is done strictly through a generic abstract-domain interface. The abstractions constructed by our techniques also expose that same interface, and thus, are compatible with existing analysis engines. As a result, our techniques are independent from specific abstractions and specific analysis engines, can be easily incorporated into existing program-analysis tools, and should be readily compatible with new abstractions to be introduced in the future.

A practical application of numeric analysis that we consider in this thesis is the automatic generation of summaries for library functions from their low-level implementation (that is, from a library's binary). The source code for library functions is typically not available. This poses a stumbling block for many source-level program analyses. Automatic generation of summary functions will both speed up and improve the accuracy of library-modeling, a process that is currently carried out by hand. This thesis addresses the automatic generation of summaries for memory-safety analysis.

Chapter 1

Introduction

The second half of the twentieth century saw the birth and the proliferation of computers from one per nation to several per household. Today, computers have become an inseparable part of our lives: we routinely entrust them with handling sensitive information, such as credit card numbers and social-security numbers; we rely on them in situations when human life is at stake, such as in airplanes' flight-control systems and in cars' anti-lock brake systems. Accordingly, ensuring that computers, and the software they execute, behave properly has become a very important problem. Simple programming errors may have grave consequences: an out-of-bounds array access may lead to someone's identity being stolen, a floating-point conversion error may lead to a spacecraft explosion (Ariane 5 crash, 1996 [40]), a division by zero may incapacitate an entire Navy ship (USS Yorktown, 1997 [36]). The problem does not stop there: even if such errors were identified and eliminated from the program code, there is a possibility that new errors will be introduced during compilation (the WYSINWYX phenomenon [10]), or due to program interaction with a library implementation that does not entirely conform to its specification. The techniques presented in this thesis are aimed at improving the applicability and precision of automatic program analysis, which is dedicated to reasoning about and understanding program behavior to detect and eliminate such errors.

1.1 Program Analysis Basics

There are many different approaches to discovering and eliminating software errors. They range from testing, in which the program's functionality is exercised by an extensive test suite, to program analysis, in which the program in never executed explicitly – instead, the code of the program is analyzed. Program analysis, itself, encompasses a wide range of techniques: from *bug-hunting* – lightweight techniques that choose to ignore some program behaviors (and, consequently, to miss some program errors) in order to achieve scalability and to generate more precise error reports; to *software verification* – heavyweight techniques that are able to guarantee the absence of errors in the program, but are much more computationally intensive. The material in this thesis is presented in the setting of software verification; however, we believe that our techniques can also benefit other branches of program analysis.

To verify that a program has no error, an analysis must systematically explore all possible program executions (we will use the term *program state* to refer to a snapshot of program execution). If the set of program states that can arise over all possible program executions (referred to as the set of *reachable states*) does not contain states that constitute errors (referred to as *error states*), the analysis may conclude that the program is correct. However, computing the set of reachable program states is by no means trivial: numeric variables may take infinitely many values[1], dynamic memory allocation precludes the analysis from knowing *a priori* the set of memory locations that must be tracked, and recursion allows program functions to be invoked in an unbounded number of contexts. In fact, the problem of computing the set of reachable program states is undecidable, in general.

Program analysis sidesteps undecidability by approximating sets of reachable states by larger sets (i.e., *supersets*) that are decidable and that can be effectively represented and manipulated by a computer. These approximations are called *abstractions*. Since abstractions over-approximate the sets of states that they represent, the analysis is able to compute an over-approximation for the set of reachable program states — and consequently, any program behavior that leads to an error state will be identified. However, these over-approximations may contain states that do not arise on any program execution. If these extra states happen to satisfy error conditions, program analysis generates spurious error reports (also known as *false positives*). The central problem of program analysis is searching for abstractions that are both computationally efficient and precise (that is,

[1] Or at least a large number of values

produce a low number of false positives). Generally, no single abstraction is able to work well in all possible situations. Thus, a large number of abstractions, each of which is tailored to a specific sets of programs and/or properties, has been designed.

Conceptually, a program analyzer consists of two components: (i) an *abstract domain*, approximates sets of reachable states and manipulates these approximations (referred to as *abstract states*) to reflect the effects of program statements; and (ii) an *analysis engine*, propagates abstract states through the program. The two components are connected through a generic interface. This modular design provides for program-analysis implementations that are parametrized by an abstract domain: as more sophisticated abstractions are designed, they can be plugged into existing program-analysis tools, as long as these new abstractions adhere to the interface. Similarly, more sophisticated analysis engines can reuse existing abstractions. Many of the techniques presented in this dissertation are positioned on this border between an abstract domain and an analysis engine, and make heavy use of the interface between the two.

Algebraically, abstract domains are best thought of as partial orders, where order is given by set containment: the smaller the set represented by an abstract state, the more precise that abstract state is. Two properties of abstract domains are of particular interest: (i) an abstract domain is said to be *distributive* if no information is lost when the abstract states computed along different program paths are combined; (ii) an abstract domain satisfies the *ascending-chain condition* if it is impossible to construct an infinite sequence of abstract states that have strictly decreasing precision. If an abstract domain possess both properties, than even the simplest analysis engine is able to compute the optimal results with respect to that abstraction: intuitively, the distributivity allows the analysis to merge paths with no penalty in precision, and the ascending-chain condition allows to handle loops effectively and precisely.

1.2 A Few Words on Terminology

The field of software verification comprises a number diverse research groups, each with its own background and its own interpretation of the common terminology. To avoid confusion, we would like to briefly explain our use of some cornerstone program-verification terms. In particular, we use the terms “sound” and “complete” in the sense of abstract-interpretation community: the sense of the terms is with respect to the set of program states; an over-approximation of the set is *sound*, but *incomplete*. In contrast, in model-checking community, these terms are often used with respect to errors reported; in this sense, static-analysis answers are complete (all true errors are reported), but error reports are unsound (reports can include false positives).

Throughout this thesis, we often use the word “precise” to characterize the quality of program-analysis results. In most cases, our use of “precise” is synonymous to “good” or “reasonable” on intuitive level. In case, there is a certain precision guarantee (e.g., a particular technique computes a *meet-over-all-valid-path (MOVP)* solution), we will state so explicitly. Also, we never use the word “imprecise” to mean *unsound*: that is, an *imprecise* solution is always an over-approximation of the optimal solution.

1.3 Numeric Program Analysis

The material presented in this dissertation is centered on *numeric program analyses*: analyses that discover numeric properties of a program. A simple example of numeric analysis is one that discovers a range of values that a variable may have at a particular program location. A more involved example is an analysis that discovers numeric relationships that hold among values of program variables, e.g., establishing that the relationship $a = 4 * b + c$ always holds among the values of variables a, b, and c at a certain program point. Such numeric properties can be used directly to identify program errors, such as out-of-bounds array accesses, integer overflow, and division by zero. While seemingly simple, such errors account for the majority of known security vulnerabilities according to CERT [19, 60, 114].

The origins of numeric program analysis date back to the early 1970s. Over the years, a rich set of numeric abstractions has been developed. These abstractions range from simple ones like *intervals*, which only keep track of upper and lower bounds for each variable; to relational ones, like *polyhedra*, which are able to establish linear relationships among variables; to automata-based *numeric decision diagrams*, which are able to represent arbitrary Presburger formulas. These abstractions exhibit varying precision/cost trade-offs and have been successfully used in practice.

However, the majority of numeric abstractions are not *distributive* and do not satisfy the *ascending-chain condition*. Thus, obtaining precise analysis results — or even reasonably precise results — remains somewhat of a black art.

Many program analyses, even those that are not directly concerned with numeric behavior of a program, often rely on numeric program-analysis techniques. To list a few examples:

- *Low-level code analysis*, such as analysis of x86 binaries, is essential for the fields of security and reverse engineering. In low-level code, typically, there are no explicit variables; rather, variables correspond to offsets from the beginning of an activation record or the beginning of the data section. Numeric operations are used to manipulate these offsets. Thus, a sophisticated numeric analysis is required just to determine which memory locations are accessed by each instruction [8].

- *Shape analysis*, an analysis that establishes properties of heap-allocated linked data structures, may use numeric quantities to represent some aspects of a shape abstraction, such as the length of a linked-list segment, or the depth of a tree [35, 117].

- *Memory-cleanness analysis*, an analysis that checks for memory-safety violations, uses numeric quantities to track the amount of memory allocated for each buffer, the offsets of pointers within the corresponding buffers, and the lengths of C-style zero-terminated strings [1, 37, 38, 107, 114].

- *Model checking*, a technique for verifying program properties, uses numeric program analysis techniques either directly (to represent numeric portions of the states of an infinite-state system [13, 20]) or indirectly (to aid predicate abstraction by strengthening the transition relation of a program [62]).

Numeric program-analysis techniques are also used in many other areas, such as the analysis of *synchronous systems* [57], real-time systems, *timed automata* [92], *hybrid systems, constraint logic programs (CLP)*, for establishing termination of Prolog programs, etc.

1.4 Thesis Contributions

In this thesis, we develop a suite of techniques with the common goal of improving the precision of numeric program analysis. The techniques address various aspects of numeric analysis, such as handling dynamically-allocated memory, dealing with programs that manipulate arrays, improving the precision of extrapolation (widening), and performing interprocedural analysis. The techniques use existing numeric abstractions as building blocks. The communication with existing abstractions is done strictly through the generic abstract-domain interface. The abstractions constructed by our techniques also expose that same interface, and thus, are compatible with existing analysis engines. The only exception to this rule is the framework of *guided static analysis* (Chapter 5), which imposes an interface on the entire program-analysis run and adheres to that interface. As the result, our techniques are independent from specific abstractions and specific analysis engines, can be easily incorporated into existing program-analysis tools, and should be readily compatible with new abstractions to be introduced in the future.

There is nothing specific about our techniques that limits their applicability only to numeric abstractions. In fact, the techniques can be applied to any abstraction, as long as that abstraction supports the required interface. However, the problems addressed by our techniques are common to numeric abstractions, and so far, we only evaluated our techniques in the setting of numeric program analysis.

1.4.1 Contributions at a Glance

This thesis makes the following contributions:

- **Summarizing abstractions [47].** We design a systematic approach for extending "standard" abstractions (that is, the abstractions that are only able to model and capture relationships among a fixed, finite set of individual program variables) with the ability to model and capture universal properties of potentially-unbounded groups of variables. Summarizing abstractions are of benefit to analyses that verify properties of systems with an unbounded number of numeric objects, such as shape analysis, or systems in which the number of numeric objects is bounded, but large.

- **Array analysis [51].** We construct an analysis that is capable of synthesizing universal properties of array elements, such as establishing that *all* array elements have been initialized (*an array kill*) and discovering constraints on the values of initialized elements. The analysis utilizes summarizing abstractions to capture and represent properties of array elements.

- **Guided static analysis [48, 49]:** We propose a framework for guiding state-space exploration performed by the analysis, and present two instantiations of the framework, which improve the precision of widening in loops with complex behavior. Widening is generally viewed as the "weakest link" of numeric analysis: ad-hoc techniques and heuristics are typically used to retain the precision of the analysis. The techniques we propose are both systematic and self-contained, and can be easily integrated into existing analysis tools.

- **Interprocedural analysis.** We investigate the use of *weighted pushdown systems (WPDSs)* as an engine for interprocedural numeric analysis. Our main numeric-program-analysis tool is implemented on top of an off-the-shelf library for WPDSs.

- **Low-level library analysis and summarization [50]:** We propose a method for constructing summary information for a library function by analyzing its low-level implementation (i.e., a library's binary). Such summary information is essential for the existing source-level program analyses to be able to analyze library calls, when the source code for the library is not available. At the heart of the method, the disassembled library code is converted into a numeric program, and the resulting program is analyzed with the use of the numeric-program-analysis techniques described above.

1.4.2 Implementation

Theoretically, all of the techniques that we propose are built around the same interface: the generic interface of an abstract domain, and it should be easy combine them within a single program-analysis tool. While this is indeed the case, due to time constraints and certain practical considerations, we chose to build two implementations, each of which implements a subset of the techniques:

- **WPDS-based analyzer:** This analyzer implements an interprocedural numeric program analysis that supports recursion, global and local variables, and *by-value* parameter passing. The implementation is based on the Parma Polyhedral Library (PPL) [7] and the WPDS++ library [69] for weighted pushdown systems. Local variables are handled with *merge functions* [75]. Guided-static-analysis techniques are used to improve widening precision. This analysis tool is used for low-level library analysis and summarization.

- **TVLA-based analyzer:** This analyzer implements an intraprocedural array analysis. The implementation is based on TVLA [78], a state-of-the-art shape-analysis tool, extended (via the use of summarizing abstractions) with the capability to model numeric properties. This implementation also uses the Parma Polyhedral Library (PPL) [7] to manipulate abstract numeric states and guided-static-analysis techniques to improve widening precision.

1.4.3 Summarizing Abstractions

Existing numeric abstractions are only able to keep track of a fixed, finite set of numeric variables. However, if a program manages memory dynamically, the set of variables that the analysis must keep track of may change as the program executes, and may not be statically bounded. For instance, keeping track of values that are stored in a linked list poses a problem to existing numeric program analyses because it is impossible to model each individual list element. A typical approach that pointer analyses use to deal with dynamically allocated memory is to partition memory locations into a fixed, finite set of groups and reason about locations in each group collectively. Partitioning can be as simple as grouping together all memory locations created at a particular allocation site, as is done by many pointer-analysis algorithms; or as complex as maintaining a *fluid* partitioning that changes during the course of the analysis, as is done by state-of-the-art shape analyses [100]. However, existing numeric abstractions cannot be used in this setting because they are incapable of such collective reasoning.

In Chapter 3, we present a framework for automatically lifting *standard* numeric abstractions to support reasoning about potentially unbounded groups of numeric variables: instead of establishing numeric properties of individual variables, lifted abstractions capture *universal* properties of groups of variables. For instance, a lifted polyhedral abstraction can capture the property that the value of each element in an array is equal to its index times two. Lifting is done by assigning a non-standard meaning to the existing abstraction. Sound and precise transformers for the lifted abstraction are automatically constructed from the transformers for the original abstraction. We used summarizing abstractions to add numeric support to TVLA, a state-of-the-art shape-analysis framework.

We collaborated with Bertrand Jeannet (IRISA, France) on distilling the ideas behind summarizing abstractions into a novel relational abstraction for functions [64, 65]; however, that work is beyond the scope of this dissertation.

1.4.4 Array Analysis

An array is a simple and efficient data structure that is heavily used. In many cases, to verify the correctness of programs that use arrays an analysis needs to be able to discover relationships among values of array elements, as well as their relationships to scalar variables. For example, in scientific programing, sparse matrices are typically represented with several arrays, and indirect indexing is used to access matrix elements. In this case, to verify that all array accesses are in bounds, an analysis has to discover upper and lower bounds on the elements stored in the index arrays. Mutual-exclusion protocols, such as the Bakery and Peterson algorithms [76, 90], use certain relationships among the values stored in a shared integer array to decide which processes may enter their critical section. To verify the correctness of these protocols, an analysis must be capable of capturing these relationships.

Static reasoning about array elements is problematic due to the unbounded nature of arrays. Array operations tend to be implemented without having a particular fixed array size in mind. Rather, the code is parametrized by scalar variables that have certain numeric relationships to the actual size of the array. The proper verification of such code requires establishing the desired property for all possible values of those parameters. These symbolic constraints on the size of the array preclude the analysis from modeling each array element as an independent scalar variable and using standard numeric-analysis techniques to verify the property. Alternatively, an entire array may be modeled as a single *summary* numeric variable. In this case, numeric properties established for this summary variable must be universally shared by all array elements. This approach, known as *array smashing* [14], resolves the unboundedness issue. However, the problem with this approach, as with any approach that uses such aggregation, is the inability to perform *strong updates* when assigning to individual array elements;[2] this can lead to significant precision loss.

In Chapter 4, we develop an analysis framework that combines *canonical abstraction* [78, 100], an abstraction that dynamically partitions memory locations into groups based on their properties, and *summarizing abstractions* [47]. The analysis uses canonical abstraction to partition an unbounded set of array elements into a bounded number of groups. Partitioning is done based on certain properties of array elements, in particular, on numeric relationships between their indices and values of scalar variables: the elements with similar properties are grouped together. Each group is represented by a single abstract array element. Summarizing numeric abstractions are used to keep track of the values and indices of array elements.

Canonical abstraction allows us to partition the set of array elements into groups, which dynamically change during the course of the analysis. For instance, if we partition array elements with respect to a loop induction variable `i` (i.e., yielding three groups of array elements: (i) the elements with indices less than the value of `i`, (ii) the element `a[i]` by itself, and (iii) the elements with indices greater than the value of `i`), then the groups of array elements, which are summarized together, change on each iteration of the loop. In particular, the indexed array element `a[i]` is always a single element in its group, which allows the analysis to perform strong updates. Also, the elements that have already been processed by the program (e.g., the ones with indices less than

[2] A strong update corresponds to a kill of a scalar variable; it represents a definite change in value to all concrete objects that the abstract object represents. Strong updates cannot generally be performed on summary objects because a (concrete) update only affects <u>one</u> of the summarized concrete objects.

the value of `i`) are kept separate from the elements that have not yet been processed (e.g., the ones with indices greater than the value of `i`), which allows the analysis to capture and maintain sharper properties for the processed array elements.

We implemented this approach to array analysis within the TVLA framework [78] and used it to analyze a number of small, but non-trivial array-manipulating programs.

1.4.5 Guided Static Analysis

Many existing numeric abstractions must rely on *extrapolation* (also referred to as *widening*) to be usable in practice. Widening attempts to guess loop invariants by observing how the program properties inferred by the analysis change during early loop iterations. Widening works well for programs with simple and easily-predictable behavior; however, as the complexity of a program increases, widening starts to lose precision. This loss of precision makes the use of widening very tricky in practice: the well-known adage goes: "If you widen without principles, you converge with no precision!" [56]. A number of *ad hoc* techniques for reclaiming lost precision have been proposed over the years. These techniques mostly rely on invariant guesses supplied by either a programmer or by a separate analysis.

In Chapter 5, we design a general framework for *guiding* the state-space exploration performed by program analysis and use instantiations of this framework to improve the precision of widening [48, 49]. The framework controls state-space exploration by applying standard program-analysis techniques to a sequence of *program restrictions*, which are modified versions of the analyzed program. The result of each standard-analysis run is used to derive the next program restriction in the sequence, and also serves as an approximation for the set of initial states used in the next analysis run. The existing program-analysis techniques are utilized "as is", making it easy to integrate the framework into existing tools. The framework is instantiated by specifying a procedure for deriving program restrictions.

To improve the precision of widening, we instantiate the framework with procedures that decompose a program with complex behavior into a sequence of simpler programs, whose complexity gradually increases. In the end, the sequence converges to the original program. Standard widening techniques are able to obtain precise results on the program restrictions that appear early in the sequence. These precise results, in turn, help obtain more precise results for later, more complex programs in the sequence, much in the same way as how successful guesses of (good) invariants help existing techniques. The two instantiations we propose in this thesis address two scenarios that are problematic to existing widening techniques:

- Loops that have multiple phases: that is, loops in which the iteration behavior changes after a certain number of iterations;
- Loops in which the behavior on each iteration is chosen non-deterministically; such loops commonly arise in the analysis of synchronous systems [46, 57] due to non-deterministic modeling of the environment (e.g., sensor signals, etc.).

As anecdotal evidence of the success of our approach, the above instantiations were able to automatically infer precise loop invariants for the two examples used by the Astrée team to motivate the use of *threshold widening*, a semi-automatic technique that relies on user-supplied thresholds (invariant guesses) [14].

1.4.6 Interprocedural Analysis

Recently, Weighted Pushdown Systems (WPDSs) emerged as an attractive engine for performing interprocedural program analysis [97]: on the one hand, WPDSs are expressive enough to capture precisely fairly complex control structure present in modern programming languages, such as Java exceptions [89]; on the other hand, WPDSs serve as a basis for higher-level analysis techniques, such as the analysis of concurrent programs [17, 21, 91]. Adopting WPDSs as an engine for numeric program analysis allows for easy integration with these analysis techniques. An additional advantage of WPDSs is the ability to answer *stack-qualified* queries, that is, the ability to determine properties that arise at a program point in a specified set of calling contexts.

In WPDSs, *weights* abstract the effect of program statements on program states. This contrasts to most existing numeric or quasi-numeric[3] analyses, in which “units of abstraction” are *sets of program states* as opposed to the *transformations of program states* (however, there are techniques for using numeric abstractions to represent program-state transformations [30, 66]). Generally, representing program-state transformations precisely is harder than representing sets of program states. However, there are definite advantages to abstracting transformations: recursive functions can be handled precisely, the same weights can be used for performing both backward and forward analyses, a single analysis run can be used to compute both a precondition for reaching a program point and the set of program states that may arise at that point (by projecting the weight computed for that program point to its domain and its range, respectively).

WPDS-based program-analysis techniques guarantee precise results for weights that are distributive and that satisfy the ascending-chain condition. Numeric abstractions, however, rarely poses either of these properties. Chapter 6, we investigate practical implications of building WPDS-based numeric program analysis: our program-analysis tool is built on top of an off-the-shelf library for WPDSs [69]. It uses *polyhedral* numeric abstraction [32, 57] implemented via the Parma Polyherdral Library [7]. The weights are constructed in a way similar to relational analysis [30, 66]: that is, a polyhedron is used to capture the relationships among the values of program variables before and after a transformation.

1.4.7 Library Summarization

Static program analysis works best when it operates on an entire program. In practice, however, this is rarely possible. For the sake of maintainability and quicker development times, software is kept modular with large parts of the program hidden in libraries. Often, commercial off-the-shelf (COTS) modules are used. The source code for COTS components and libraries (such as Windows dynamically linked libraries) is not usually available. In practice, source-level analysis tools resort to either of the following two techniques:

[3] That is, program analyses in which only part of the abstraction is numeric.

- abstract transformers for a set of *important* library calls (such as *memcpy* and *strlen*) are hardcoded into the analysis; the remaining calls are treated either conservatively (e.g., by assuming that any part of memory can be affected) or unsoundly (e.g., by assuming that the function call does not affect the state of the program);

- a collection of hand-written source-code stubs that emulate some aspects of library code is used by the analysis as a model of the effects of calls on library entry points.

Both approaches are less than ideal. The former approach is not extensible: modeling of new library functions necessitates changes to the analysis. The latter approach provides the means for modeling new libraries without changing the analysis; however, library-function stubs have to be created manually — a process that is both error-prone and tedious.

An attractive goal for program analysis is to derive automatically the *summaries* for library calls by analyzing the low-level implementation of the library (e.g., the library's binary). Such summaries should consist of a set of assertions (*error triggers*) that must be checked at the call-site of a library function to ensure that no run-time errors may occur during the invocation of the function, and a program-state transformer that specifies how to map the program state at the function call-site into the program state at the corresponding return-site.

We believe that there are certain benefits to constructing the models of library calls from the low-level implementation of the library, as opposed to doing it from the high-level specifications. In particular,

- formal library specifications are hard to get hold of, while low-level implementations are readily available;

- even if a formal specification is available, there is no easy way to verify that a particular library implementation conforms to the specification;

- the analysis of an actual library implementation may uncover bugs and undesirable features in the library itself.

In this thesis, we take the first steps towards automatically constructing summaries for library functions. We design a tool that constructs library-function summaries for memory-safety analysis. The tool works directly on the library implementations (x86 code, at the moment):

- First, CodeSurfer/x86, an off-the-shelf tool for binary-code analysis, is used to perform initial analysis of the binary and to construct an *intermediate representation (IR)* of the function;

- The IR is used to generate a numeric program that captures the memory-related behavior of the function: auxiliary numeric variables are associated with each pointer variable to keep track of its allocation bounds in the spirit of other memory-safety analyses [37, 38]. At memory dereferences, the allocation bounds of the the dereferenced pointer variable are checked explicitly, and in case of an error, the control is transferred to specially-inserted error points.

- The numeric program, generated in the previous step, is analyzed with an off-the-shelf numeric program-analysis tool. We used our WPDS-based analyzer, which was described in the previous section.

- The numeric-analysis results are used to generate error assertions and program-state transformers. At this step, we get extra millage from the WPDS-based analysis tool: weights computed by the analysis for the set of return statements are used directly as the program-state transformers; error triggers are obtained by computing preconditions for reaching the error points.

In this work, we concentrated primarily on the issues that are of immediate relevance to numeric program analysis: i.e., the generation and the analysis of a numeric program. A number of problems that must be addressed to make our method usable in practice still remain open. For instance, our tool produces an error trigger for each memory access in the function (which may number in the hundreds). Many of these error triggers are redundant; others could be consolidated into a single, simpler error condition; hoverer, techniques for performing such consolidation still need to be developed.

1.5 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 introduces *numeric program analysis*: it describes commonly used numeric abstractions, and shows how to instantiate these analyses in the framework of abstract interpretation. Chapter 3 presents the framework for extending *standard* numeric abstractions with the capability to express universal properties for groups of numeric quantities. Chapter 4 uses the framework of Chapter 3 to define a numeric analysis that is able to synthesize universal properties of array elements. Chapter 5 presents a general framework for guiding the analysis through the state-space of the analyzed program: two instantiations of the framework, which improve the precision of extrapolation, are presented. Chapter 6 addresses the use of WPDSs as an engine for numeric program analysis. Chapter 7 discusses a particular application of numeric program analysis: analyzing low-level library implementations to create summaries for library functions.

Chapter 2

Numeric Program Analysis

This chapter presents the foundations of numeric program analysis. Of particular importance is the notion of an *abstract domain*: the techniques in Chapters 3, 4, and 5 use generic abstract domains as basic building blocks. We describe how to instantiate a simple intraprocedural numeric program analysis within the framework of abstract interpretation [27, 29]. For further reading we suggest an excellent in-depth coverage of numeric program analysis by Mine [87, Chapter 2]. Additional background material and definitions that are specific to particular techniques will be given in the corresponding chapters.

2.1 Numeric Programs

A program is specified by a *control flow graph (CFG)* — i.e., a directed graph $G = (V, E)$, where V is a set of program locations, and $E \subseteq V \times V$ is a set of edges that represent the flow of control. A node $n_e \in V$ denotes a unique entry point into the program. Node n_e is not allowed to have predecessors.

The state of a program is often modeled using a fixed, finite set of variables, *Vars*, whose values range over a set $\mathbb{V}$. We assume that the set $\mathbb{V}$ is application specific, e.g., $\mathbb{V}$ can be the set of integers ($\mathbb{Z}$), the set of rationals ($\mathbb{Q}$), or the set of reals ($\mathbb{R}$). We will use $\mathbb{B}$ to denote the set of Boolean values, i.e., $\mathbb{B} = \{\textit{true},\ \textit{false}\}$.

A *program state* S is a function that maps each program variable to a corresponding value, i.e., $S : \textit{Vars} \rightarrow \mathbb{V}$. We will use $\Sigma = \textit{Vars} \rightarrow \mathbb{V}$ to denote the set of all possible program states.

2.1.1 Numerical Expressions and Conditionals

We do not impose any constraints on the numeric expressions and conditionals that can be used in the program: that is $x + y$, $\sqrt[3]{y}$, x **mod** $5 = 0$, and $\sin^2(x) + \cos^2(x) = 1$ all represent valid numeric expressions and conditionals. The semantics of the expressions and conditionals is defined by a family of functions $[\![\cdot]\!]$, which map the values assigned to the free variables in the expression (conditional) to the result of the expression (conditional). That is, $[\![x + y]\!](3, 5)$ yields 8 and $[\![x \textbf{ mod } 5 = 0]\!](25)$ yields *true*. More formally, let Φ denote the set of all valid numeric expressions, and let $\phi(v_1, \dots, v_k) \in \Phi$ be a k-ary expression. Then, the semantics of ϕ is given by a function

$$[\![\phi(v_1, \dots, v_k)]\!] : \mathbb{V}^k \to \mathbb{V}.$$

Similarly, let Ψ denote the set of all valid numeric conditionals, and let $\psi(v_1, \dots, v_k) \in \Psi$ be a k-ary conditional. Then the semantics of ψ is given by a function

$$[\![\psi(v_1, \dots, v_k)]\!] : \mathbb{V}^k \to \mathbb{B}.$$

2.1.2 Support for Nondeterminism

Sometimes it is convenient to be able to specify a certain degree of non-determinism in the program. For instance, nondeterminism can be used to model the effects of the environment, e.g., to model user input. We support nondeterminism by allowing a special constant '$?$' to be used within an expression: '$?$' chooses a value from $\mathbb{V}$ nondeterministically. To accommodate '$?$', we lift the semantics of expressions and conditionals to return sets of values: e.g., $[\![x{+}?]\!]_{ND}$ yields $\mathbb{V}$ for any value of x, and $[\![? \textbf{ mod } 3]\!]_{ND}$ yields the set $\{0, 1, 2\}$.

Without loss of generality, let expression $\phi(v_1, \dots, v_k)$ have r occurrences of '$?$' in it. Let $\tilde{\phi}(v_1, \dots, v_k, w_1, \dots, w_r)$ denote an expression obtained by substituting each occurrence of '$?$' in ϕ with a fresh variable $w_i \notin$ *Vars*. Then, the semantics for ϕ is defined as follows (let $\bar{\alpha} \in \mathbb{V}^k$):

$$[\![\phi(v_1, \dots, v_k)]\!]_{ND}(\bar{\alpha}) = \left\{ [\![\tilde{\phi}(v_1, \dots, v_k, w_1, \dots, w_r)]\!](\bar{\alpha}, \bar{\beta}) \mid \bar{\beta} \in \mathbb{V}^r \right\}$$

The nondeterministic semantics for conditionals is defined similarly.

2.1.3 Evaluation of Expressions and Conditionals

We define the semantics for evaluating expressions and conditionals in a program state in a straightforward way. Let $S \in \Sigma$ denote an arbitrary program state, and let $\phi(v_1, \ldots, v_k) \in \Phi$, where $v_i \in \mathit{Vars}$, be an expression. The function $[\![\phi(v_1, \ldots, v_k)]\!] : \Sigma \to \wp(\mathbb{V})$ is defined as follows:

$$[\![\phi(v_1, \ldots, v_k)]\!](S) = [\![\phi(v_1, \ldots, v_k)]\!]_{ND}(S(v_1), \ldots, S(v_k)).$$

Similarly, let $\psi(v_1, \ldots, v_k) \in \Psi$, where $v_i \in \mathit{Vars}$, be a conditional. The function $[\![\psi(v_1, \ldots, v_k)]\!] : \Sigma \to \wp(\mathbb{B})$ is defined as follows:

$$[\![\psi(v_1, \ldots, v_k)]\!](S) = [\![\psi(v_1, \ldots, v_k)]\!]_{ND}(S(v_1), \ldots, S(v_k)).$$

From now on, we will omit the lists of free variables when referring to expressions and conditionals, unless those lists are important to the discussion.

2.1.4 Concrete Semantics of a Program

The function $\Pi_G : E \to (\Sigma \to \Sigma)$ assigns to each edge in the CFG the concrete semantics of the corresponding program-state transition. Two types of transitions are allowed:

- **Assignment transition, $\bar{x} \leftarrow \bar{\phi}$:** An assignment transition allows multiple variables to be updated in parallel; i.e., $\bar{x} \in \mathit{Vars}^m$, where $1 \le m \le |\mathit{Vars}|$, with an additional constraint that each variable may appear at most once in $\bar{x}$. Also, $\bar{\phi} \in \Psi^m$. As an example, the assignment transition $\langle x, y \rangle \leftarrow \langle y, x+1 \rangle$ assigns the value of variable y to variable x, and the value of $x+1$ to variable y. The semantics of an assignment transition is defined as follows (we use $x[i]$ to denote the i-th component of vector $\bar{x}$):

  $$[\![\bar{x} \leftarrow \bar{\phi}]\!](S) = \left\{ S' \in \Sigma \;\middle|\; \forall v \in \mathit{Vars} \left[\begin{array}{ll} S'(v) \in [\![\phi[i]]\!]_{ND}(S) & \text{if } v = x[i]\,,\ i \in [1, m] \\ S'(v) = S(v) & \text{otherwise} \end{array} \right] \right\}$$

  Typically, only a single variable is updated by an assignment transition. In that case, we will omit the vector notation, e.g., $x \leftarrow x+1$. The assignment transition $x \leftarrow ?$ “forgets” the value of variable x.

- **Assume transition, *assume*(ψ):** An assume transition filters out program states in which the condition $\psi \in \Psi$ does not hold. For uniformity, we define the semantics of the transition to map a program state to a singleton set containing that program state, if the condition ψ holds in that state; otherwise, a program state is mapped to the empty set

$$[\![\textit{assume}(\psi)]\!](S) = \begin{cases} \{S\} & \text{if } \textit{true} \in [\![\psi]\!]_{ND}(S) \\ \emptyset & \text{otherwise} \end{cases}$$

The semantics of program-state transitions is extended trivially to operate on sets of program states

$$\Pi_G(e)(\textit{SS}) = \bigcup_{S \in \textit{SS}} \Pi_G(e)(S),$$

where $e \in E$ and $\textit{SS} \subseteq \Sigma$.

2.1.5 Collecting Semantics of a Program

We will use maps $\Theta : V \to \wp(\Sigma)$ from program locations to program states to collect the sets of reachable states. Let $\Theta_\triangleright$ denote a map that represents the initial state of the program. Typically, we assume that the program execution starts at the entry point n_e, and that program variables are not initialized:

$$\Theta_\triangleright(v) = \begin{cases} \Sigma & \text{if } v = n_e \\ \emptyset & \text{otherwise} \end{cases}, \quad \text{for all } v \in V$$

The *collecting semantics* of a program (that is, a function that maps each program location to the set of program states that arise at that location), is given by the least map $\Theta_\star$ that satisfies the following conditions:

$$\Theta_\star(v) \supseteq \Theta_\triangleright(v), \quad \text{and} \quad \Theta_\star(v) = \bigcup_{\langle u,v\rangle \in E} \Pi_G(\langle u,v\rangle)(\Theta_\star(u)), \text{ for all } v \in V \tag{2.1}$$

The goal of program analysis is, given $\Theta_\triangleright$, compute $\Theta_\star$. However, this problem is generally undecidable.

2.1.6 Textual Representation of a Program

Our primary view of the program is that of a control flow graph in which nodes correspond to program locations and edges are annotated with program-state transitions. In fact, the input languages for the program-analysis tools that we have implemented also maintain this view of programs. However, for the sake of readability, we present the programs that appear in various part of this thesis in a C-like language. The language supports simple control structures, such as `if` statements, `while` statements, and `goto` statements. These control structures are converted into a CFG in straightforward way. The statement `assert(`ψ`)` corresponds to an `if(`ψ`)` statement in which the *else*-clause transfers the control to an *error node*: a unique error node is created for each `assert` statement. Error nodes are sink nodes, i.e., they have no successors. The statement "`if(*)`" chooses control non-deterministically, i.e., the transition $assume(true)$ is used to transfer control to both the *then*-clause and *else*-clause. Whenever possible, we show the program's CFG alongside the textual representation of the program.

2.2 Program Analysis

Program analysis sidesteps undecidability by using abstraction: sets of program states are over-approximated by elements of an abstract domain $\mathbb{D} = \langle D, \sqsubseteq, \top, \bot, \sqcup, \sqcap \rangle$, where $\sqsubseteq$ is a binary relation that is reflexive, transitive, and anti-symmetric: it imposes a partial order on D; $\top$ and $\bot$ denote, respectively, the greatest and the least elements of D with respect to $\sqsubseteq$; $\sqcup$ and $\sqcap$ denote the least upper bound (*join*) operator and the greatest lower bound (*meet*) operator, respectively.

The elements of $\mathbb{D}$ are linked to sets of concrete program states by a pair of functions $\langle \alpha, \gamma \rangle$, where $\alpha : \wp(\Sigma) \to D$ is an *abstraction function*, which constructs an approximation for a set of states, and $\gamma : D \to \wp(\Sigma)$ is a *concretization function*, which gives meaning to domain elements; The functions α and γ are chosen to form a Galois connection, that is

$$\forall S \in \wp(\Sigma)\ \forall d \in D\ \ [\ \alpha(S) \sqsubseteq d\ \Leftrightarrow\ S \subseteq \gamma(d)\ ]$$

It follows immediately that $d_1 \sqsubseteq d_2 \Rightarrow \gamma(d_1) \subseteq \gamma(d_2)$; thus, the smaller the abstract-domain element with respect to $\sqsubseteq$, the smaller is the set of program states it represents. The least element, $\bot$, typically represents the empty set, i.e., $\gamma(\bot) = \emptyset$. The greatest element, $\top$, represents the entire set of program states, i.e., $\gamma(\top) = \Sigma$. The join operator and the meet operator soundly approximate set union and set intersection, respectively.

2.2.1 Abstract Semantics of a Program

To perform program analysis, the program-state transitions that are associated with the edges of a control flow graph also need to be abstracted. We will use the map $\Pi_G^\sharp : E \to (D \to D)$ to specify corresponding abstract transformers for each edge in the CFG. We say that $\Pi_G^\sharp$ is a *sound approximation* of Π_G if the following condition holds:

$$\forall e \in E \;\; \forall d \in D \;\left[\; \Pi_G(e)(\gamma(d)) \;\subseteq\; \gamma(\Pi_G^\sharp(e)(d)) \;\right].$$

Also, for a program-state transition τ, we say that its abstraction $\tau^\sharp$ is the *best* abstract transformer, if $\tau^\sharp = \alpha \circ \tau \circ \gamma$; and we say that $\tau^\sharp$ is the *exact* abstract transformer, if $\gamma \circ \tau^\sharp = \tau \circ \gamma$.

2.2.2 Abstract Collecting Semantics

To refer to abstract states at multiple program locations, we define *abstract-state maps* $\Theta^\sharp : V \to D$. We also define the operations α, γ, $\sqsubseteq$, and $\sqcup$ for $\Theta^\sharp$ as point-wise extensions of the corresponding operations for the abstract domain $\mathbb{D}$.

Program analysis computes a sound approximation for the set of program states that are reachable from $\Theta_\triangleright$. Typically, the result of program analysis is an abstract-state map $\Theta_\star^\sharp$ that satisfies the following property

$$\forall v \in V : \left[\Theta_\triangleright^\sharp(v) \sqcup \bigsqcup_{\langle u,v\rangle \in E} \Pi_G^\sharp(\langle u,v\rangle)(\Theta_\star^\sharp(u))\right] \sqsubseteq \Theta_\star^\sharp(v), \tag{2.2}$$

where $\Theta_\triangleright^\sharp = \alpha(\Theta_\triangleright)$ is the approximation for the set of initial states of the program. It follows trivially from the definition that the resulting approximation is sound, that is $\Theta_\star(v) \subseteq \gamma(\Theta_\star^\sharp(v))$ for all $v \in V$.

2.3 Iterative Computation

This section explains one methodology for computing $\Theta_\star^\sharp$, the abstract collecting semantics of a program, by means of an iterative process of successive approximation.

2.3.1 Kleene Iteration

If the abstract domain $\mathbb{D}$ and the set of abstract transformers in $\Pi_G^\sharp$ possess certain algebraic properties, then $\Theta_\star^\sharp$ can be obtained by computing the following sequence of abstract-state maps until it stabilizes:

$$\Theta_0^\sharp = \Theta_\triangleright^\sharp \quad \text{and} \quad \Theta_{i+1}^\sharp(v) = \bigsqcup_{\langle u,v\rangle \in E} \Pi_G^\sharp(\langle u,v\rangle)(\Theta_i^\sharp(u)) \tag{2.3}$$

In particular, the abstract transformers in $\Pi_G^\sharp$ must be *monotone*, i.e.,

$$\forall e \in E \;\; \forall d_1, d_2 \in D \; \Big[\, d_1 \sqsubseteq d_2 \;\; \Rightarrow \;\; \Pi_G^\sharp(e)(d_1) \sqsubseteq \Pi_G^\sharp(e)(d_2) \, \Big] .$$

To ensure termination, the abstract domain $\mathbb{D}$ must satisfy the *ascending-chain condition*; i.e., every sequence of elements $(d_k) \in D$ such that $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \ldots$ must eventually become stationary.

Additionally, if the transformers in $\Pi_G^\sharp$ *distribute* over the join operator of $\mathbb{D}$, i.e., if

$$\forall e \in E \;\; \forall d_1, d_2 \in D \; \Big[\, \Pi_G^\sharp(e)(d_1 \sqcup d_2) \;\; = \;\; \Pi_G^\sharp(e)(d_1) \sqcup \Pi_G^\sharp(e)(d_2) \, \Big] ,$$

then the solution obtained for $\Theta_\star^\sharp$ is the most precise (i.e., least) solution that satisfies Eqn. (2.2).

2.3.2 Widening

If the domain does not satisfy the ascending-chain condition, then the sequence defined in Eqn. (2.3) may not necessarily converge. To make use of such domains in practice, an extrapolation operator, called *widening*, is defined. A widening operator (∇_k) must possess the following properties:

- For all $d_1, d_2 \in D$, for all $i \geq 0$, $d_1, d_2 \sqsubseteq d_1 \nabla_i d_2$.
- For any ascending chain $(d_k) \in D$, the chain defined by $d_0' = d_0$ and $d_{i+1}' = d_i' \nabla_i d_{i+1}$ is not strictly increasing.

To make an iterative computation converge, a set $W \subseteq V$ of widening points is identified: W must be chosen in such a way that every loop in Π is *cut* by a node in W. Typically, W is chosen to contain the heads of all of the loops in the program. The iteration proceeds as follows:

$$\Theta_0^\sharp = \Theta_\triangleright^\sharp \quad \text{and} \quad \Theta_{i+1}^\sharp(v) = \Theta_i^\sharp(v) \ \bowtie \bigsqcup_{\langle u,v\rangle \in E} \Pi_G^\sharp(\langle u,v\rangle)(\Theta_i^\sharp(u)) \tag{2.4}$$

where $\bowtie$ is ∇_i if $v \in W$ and $\sqcup$, otherwise.

Intuitively, widening attempts to guess loop invariants by observing program states that arise on the first few iterations of the loop. Typically, delaying the application of widening for a few iterations tends to increase the precision of the analysis. To delay the application of widening for k iterations, the widening operator can be redefined as follows:

$$\nabla_i^{[k]} = \begin{cases} \sqcup & \text{if } i < k \\ \nabla_{i-k} & \text{otherwise} \end{cases}$$

Often the definition of the widening operator is independent from the iteration number on which the operator is invoked. In that case, we denote the widening operator by ∇, with no subscripts.

2.3.3 Narrowing

The limit of the sequence in Eqn. (2.4), which we will denote by $\Theta_\triangleleft^\sharp$, is sound, but generally overly conservative. It can be refined to a more precise solution by computing a *descending-iteration sequence*:

$$\Theta_0^\sharp = \Theta_\triangleleft^\sharp \quad \text{and} \quad \Theta_{i+1}^\sharp(v) = \bigsqcup_{\langle u,v\rangle \in E} \Pi_G^\sharp(\langle u,v\rangle)(\Theta_i^\sharp(u)) \tag{2.5}$$

However, for this sequence to converge, the abstract domain must satisfy the *descending-chain condition*, that is, the domain must contain no infinite strictly-decreasing chains. If the abstract domain does not satisfy this property, convergence may be enforced with the use of a *narrowing operator*. A narrowing operator, (Δ_k) must possess the following properties:

- For all $d_1, d_2 \in D$, for all $i \geq 0$, $\quad d_2 \sqsubseteq d_1 \Rightarrow [\ d_2 \sqsubseteq d_1 \, \Delta_i \, d_2 \ \wedge \ d_1 \, \Delta_i \, d_2 \sqsubseteq d_1\ ]$.
- For any descending chain $(d_k) \in D$, the chain defined by $d'_0 = d_0$ and $d'_{i+1} = d'_i \, \Delta_i \, d_{i+1}$ is not strictly decreasing.

The computation of the descending sequence proceeds as follows:

$$\Theta_0^\sharp = \Theta_\triangleleft^\sharp \quad \text{and} \quad \Theta_{i+1}^\sharp(v) = \Theta_i^\sharp(v) \ \bowtie \bigsqcup_{\langle u,v\rangle \in E} \Pi_G^\sharp(\langle u,v\rangle)(\Theta_i^\sharp(u)) \tag{2.6}$$

where $\bowtie$ is Δ_i if $v \in W$ and $\sqcap$, otherwise.

The overall result of the analysis, $\Theta_\star^\sharp$, is the limit of the sequence computed in accordance with Eqn. (2.6). Typically, $\Theta_\star^\sharp$ is not the most precise abstraction of the program's collecting semantics with respect to the property in Eqn. (2.2). An important thing to note is that with standard abstract-interpretation methods the computation of the *ascending sequence* (Eqn. (2.4)) for the entire program must precede the computation of the *descending sequence* (Eqn. (2.6)) for the analysis to converge. The techniques that we developed — and present in Chapter 5 — allow the ascending and descending computations to be interleaved, while still guaranteeing that the analysis converges. Moreover, the techniques from Chapter 5 generally give more precise results than standard techniques (see §5.7).

Meaningful narrowing operators are much harder to define than widening operators; thus, many abstract domains do not provide them. For those domains, the descending-iteration sequence from Eqn. (2.5) is, typically, truncated after some fixed number of iterations. One way to truncate the iteration sequence is to define a simple domain-independent narrowing operator that "cuts off" the decreasing sequence after some number of iteration; i.e., to truncate after k iterations, the narrowing operator can be defined as follows:

$$d_1 \, \Delta_i^{[k]} \, d_2 = \begin{cases} d_2 & \text{if } i < k \\ d_1 & \text{otherwise} \end{cases}$$

2.3.4 Chaotic Iteration

Computing the sequence of abstract-state maps according to Eqns. (2.4) and (2.6) is not efficient in practice: on each step, the mappings for *all* program points are recomputed even though only a small number of them actually change value. *Chaotic iteration* allows one to speed up the computation by only updating a value at single program point on each iteration of the analysis: given a *fair* sequence of program points $\sigma \in V^\infty$ (that is, a sequence in which each program point appears infinitely often), the ascending-iteration sequence can be computed as follows:

$$\Theta^\sharp_{i+1}(v) = \begin{cases} \Theta^\sharp_i(v) \nabla_i \bigsqcup_{\langle u,v\rangle \in E} \Pi^\sharp_G(\langle u,v\rangle)(\Theta^\sharp_i(u)) & \text{if } v = \sigma[i] \text{ and } v \in W \\ \Theta^\sharp_i(v) \sqcup \bigsqcup_{\langle u,v\rangle \in E} \Pi^\sharp_G(\langle u,v\rangle)(\Theta^\sharp_i(u)) & \text{if } v = \sigma[i] \text{ and } v \notin W \\ \Theta^\sharp_i(v) & \text{otherwise } (v \neq \sigma[i]) \end{cases} \tag{2.7}$$

The descending iteration sequence is computed in a similar way, with ∇ replaced with Δ, and $\sqcup$ replaced with $\sqcap$.

The use of chaotic iteration raises the question of an effective *iteration strategy* — that is, an order in which program points are visited that minimizes the amount of work that the analysis performs. This problem was addressed by Bourdoncle [18]. He proposed a number of successful iteration strategies based on the idea of using a *weak topological order (WTO)* of the nodes in the program's control-flow graph. A WTO is a hierarchical structure that decomposes *strongly-connected components (SCCs)* in the graph into a set of nested WTO components. (In structured programs, WTO components correspond to the loops in the program.) Of particular interest is the *recursive* iteration strategy.

Definition 2.1 (Recursive Iteration Strategy [18]) The recursive iteration strategy recursively stabilizes the sub-components of a WTO component before stabilizing that WTO component.

Intuitively, the recursive iteration strategy forces the analysis to stay within a loop until the stable solution for that loop is obtained. Only then the analysis is allowed to return to the outer loop. The recursive iteration strategy has the property that one only needs to check for stabilization at the head of the corresponding WTO component (intuitively, only at the head of the loop).

Theorem 2.2 (Recursive iteration stabilization [18, Theorem 5]) For the recursive iteration strategy, the stabilization of a WTO component can be detected by the stabilization of its head.

In Chapter 5, we make use of this property to guarantee the convergence of the *lookahead-widening* technique.

2.4 Abstract Domain Interface

The interface of an abstract domain consists of the domain operations (such as $\sqsubseteq$ and $\sqcup$) and the abstract transformers in $\Pi_G^\sharp$. Typically, it is not the case that the transformer for each state transition must be manually constructed. Rather, the abstract domain provides a general scheme for constructing classes of abstract transformers, i.e., a scheme for constructing abstract transformers for arbitrary assignment transitions and arbitrary assume transitions. In this thesis, we assume that abstract domains are equipped with abstract transformers for any possible assignment transitions and assume transitions.[1]

The functions α and γ are mostly theoretical devices; thus, we do not view them as part of the interface. In particular, it is rarely computationally feasible to use γ directly. However, sometimes we rely on the *symbolic-concretization* function $\hat{\gamma}$ [96], which represents the meaning of an abstract domain element as a formula in some suitable logic. The α function typically would be used exactly once, to compute the abstract domain element for the initial state; in practice, it is rare to implement α as an explicit function in an analyzer (the TVLA system [78] is one exception).

Formally, to be compatible with our techniques, an abstract domain must provide the following set of operations:

- the comparison operator ($\sqsubseteq$);
- the join operator ($\sqcup$), and the meet operator ($\sqcap$);
- the widening operator (∇), and (*optionally*) the narrowing operator (Δ);
- the constructors for the greatest element ($\top$), and the smallest element ($\bot$);
- the abstract transformers for arbitrary assignment transitions ($[\![\bar{x} \leftarrow \bar{\phi}]\!]^\sharp$) and for arbitrary assume transitions ($[\![\mathit{assume}(\psi)]\!]^\sharp$);
- (*Optionally*) the symbolic-concretization function ($\hat{\gamma}$);

[1] In case an expression or conditional are beyond the capabilities of the abstract domain, the abstract transformers err on the safe (sound) side by returning $\top$.

2.5 Numeric Abstractions

The focus of this thesis is numeric program analysis. Below, we briefly overview existing numeric abstractions (that is, particular instantiations of the abstract-domain interface), and take a detailed look at the *polyhedral* abstraction, which we use in the implementations of our tools and to illustrate the techniques presented in this thesis.

Numeric analyses have been a research topic for several decades, and a number of numeric domains that allow to approximate the numeric state of a program have been designed over the years. These domains exhibit varying precision/cost trade-offs, and target different types of numeric properties. However, all numeric domains share the same view of "the world": each program state $S : \mathit{Vars} \rightarrow \mathbb{V}$ is viewed as a point in a multi-dimensional space $\mathbb{V}^n$, where $n = |\mathit{Vars}|$. Intuitively, each variable is associated with a dimension of a multi-dimensional space; the coordinate of the point along that dimension represents the value of the variable in the corresponding program state. Sets of program states correspond to subsets of the space $\mathbb{V}^n$. Numeric abstract domains represent and manipulate these subsets with varying degrees of precision.

Tab. 2.1 lists a number of existing abstract domains, along with the numeric properties those domains are able to represent. In terms of precision, numeric abstract domains can be categorized into three groups:

- **Non-relational domains:** These domains are only able to represent properties of individual variables, e.g., the range of values that a variable may hold. They cannot represent relationships among variables. However, these domains are very efficient in practice and allow scalable analyses to be constructed.

- **Relational domains:** These domains are able to represent arbitrary relationships (of a certain class) among variables: e.g., polyhedra are capable of representing arbitrary *linear* relationships. However, these domains have poor computational complexity and do not scale well in practice.

- **Weakly-relational domains:** These domains attempt to strike a compromise between relational and non-relational abstract domains by limiting in some way the kind of numeric

	Name	Constraints	Bibliography
Non-Relational	Constant Propagation	$v_i = \alpha_i$	Kildall [70]
	Signs	$\pm v_i \geq 0$	Cousot et al. [28]
	Intervals	$\alpha_i \leq v_i \leq \beta_i$	Cousot et al. [28]
	Simple Congruences	$v_i \equiv \alpha_i$ **mod** β_i	Granger [52, 53]
	Interval Congruences	$v_i \in [\alpha_i, \beta_i]$ **mod** γ_i	Masdupuy [82]
Weakly-Relational	Zones	$v_i - v_j \leq \alpha_{ij}$	Miné [84]
	Octagons	$\pm v_i \pm v_j \leq \alpha_{ij}$	Miné [85]
	Zone Congruences	$v_i - v_j \equiv \alpha_{ij}$ **mod** β_{ij}	Miné [86]
	TVPLI	$\alpha_{ij} \cdot v_i + \beta_{ij} \cdot v_j \leq \gamma_{ij}$	Simon et al. [109]
	Octahedra	$\sum_i \alpha_{ij} \cdot v_i \leq \beta_j$ $\alpha_{ij} \in \{-1, 0, 1\},\ v_i > 0$	Clarisó et al. [23, 24]
	TCM	$\sum_i \alpha_{ij} \cdot v_i \leq \beta_j$ α_{ij} fixed throughout the analysis	Sankaranarayanan et al. [103]
Relational	Liner Equalities	$\sum_i \alpha_{ij} \cdot v_i = \beta_j$	Karr [68]
	Linear Congruences	$\sum_i \alpha_{ij} \cdot v_i \equiv \beta_j$ **mod** γ_j	Granger [52]
	Polyhedra	$\sum_i \alpha_{ij} \cdot v_i \leq \beta_j$	Cousot et al. [32]
	Trapezoidal Congruences	$\sum_i \alpha_{ij} \cdot v_i \in [\beta_j, \gamma_j]$ **mod** δ_j	Masdupuy [80]
	Arithmetic automata	Arbitrary Presburger Formulas	Bartzis et al. [12, 13]

Table 2.1 A list of existing numeric domains.

relationships they can represent. Some domains, like TVPLI[2] [109], restrict the number of variables that can participate in a relationship. Others, like octahedra [23, 24], restrict the values of coefficients that can occur in the relationships. Some, like octagons [85], restrict both.

[2] Two variables per linear inequality

Numeric abstract domains also vary in terms of implementation. Some domains, like TCM [103] and TVPLI [109], rely on linear-programming techniques. Others, like zones [84], octagons [85], and zone congruences [86], are graph-based (in fact, they operate on a matrix representation of a graph). One domain, arithmetic automata [12, 13], uses a finite-state automaton to recognize binary representations of integer values. Below, we take a detailed look at the implementation of the polyhedral domain [32, 57].

2.5.1 The Polyhedral Abstract Domain

The polyhedral abstract domain was introduced almost 30 years ago by Cousot and Halbwachs [32]. Since then, a number of implementations of the domain have been developed, e.g., the New Polka library [63] and the Parma Polyhedral Library [7].

Representation. The implementation of the domain is based on a dual representation of polyhedra. A polyhedron $P \subseteq \mathbb{R}^n$ for some positive integer n can be represented in two distinct ways:

- *Constraint representation.* A polyhedron is represented as the intersection of a finite set of linear constraints C, where each constraint $\langle \bar{c}, b \rangle \in C$, where $\bar{c} \in \mathbb{R}^n$ and $b \in \mathbb{R}$, defines a half space of $\mathbb{R}^n$ as follows $\{\bar{x} \in \mathbb{R}^n \mid \bar{c} \cdot \bar{x} \leq b\}$. The following polyhedron is represented by the constraint system C:

$$P = con(C) = \{\bar{p} \in \mathbb{R}^n \mid \forall \langle \bar{c}, b \rangle \in C \ \ [\bar{c} \cdot \bar{p} \leq b]\}.$$

- *Frame representation.* A polyhedron is represented by a system of *generators*: a finite set of *points* $Q \subset \mathbb{R}^n$ and a finite set of *rays* $R \subset \mathbb{R}^n$. Let $|Q| = q$ and let $|R| = r$. The following polyhedron is represented by a system of generators $\langle Q, R \rangle$:

$$P = gen(\langle Q, R \rangle) = \left\{ R\rho \ + Q\pi \in \mathbb{R}^n \ \mid \ \rho \in \mathbb{R}^r_+, \ \pi \in \mathbb{R}^q_+, \text{ and } \sum_{i=1}^{q} \pi[i] = 1 \right\},$$

  where $\mathbb{R}_+$ denotes the set of non-negative reals.

Some operations can be performed more efficiently on the constraint representation; others can be performed more efficiently on the generator representation. In practice, implementations maintain both representations: the representations are synchronized by converting the information in the up-to-date representation to the out-of-date representation. These conversions are computationally expensive and constitute a major scalability problem for the domain.

Abstract Transformers. In the following, let P_1 and P_2 denote two elements of the polyhedral abstract domain.

- *Comparison operator:* The comparison operator uses both representations. To decide whether $P_1 \sqsubseteq P_2$ holds, the elements of the frame of P_1 are checked against the constraints of P_2: if every element of the frame is subsumed, the relationship holds.

- *The join and meet operators:* The join operator utilizes the generator representation: to compute $P_1 \sqcup P_2$, the union of their generator representations is taken. The meet operator relies on the constraint representation: $P_1 \sqcap P_2$ is computed as the union of the constraint representations of P_1 and P_2. Both of these operations may cause redundancy in the respective representation. The redundancy is typically eliminated by a conversion to the other representation and back.

- *Widening operator.* The widening operator uses both representations. Intuitively, to compute $P_1 \nabla P_2$ one needs to remove from P_1 the constraints that are not satisfied by P_2. The implementation, however, is much more tricky: the result of widening P_1 with P_2 is a polyhedron that includes all of the constraints of P_2 that are saturated by the same set of frame elements of P_1 as some constraint of P_1 [6]. The necessary condition for this to work is that $P_1 \sqsubseteq P_2$.

- *The $\top$ and $\bot$ elements.* The $\top$ element is a polyhedron with an empty constraint system. The $\bot$ element is a polyhedron whose constraint system contains an infeasible constraint, e.g., $1 \leq 0$.

- *Abstract transformers.* Abstract transformers for polyhedra are only able to handle linear expressions and linear conditionals. The $[\![\bar{x} \leftarrow \bar{\phi}]\!]^\sharp(P_1)$ transformer is computed by translating each generator of P_1 according to the assignment transition $\bar{x} \leftarrow \bar{\phi}$. The $[\![\mathit{assume}(\psi)]\!]^\sharp(P_1)$

transformer is computed by adding the linear constraint ψ to the constraint representation of P_1. Non-linear expressions are approximated by returning $\top$; non-linear conditionals are approximated by returning the original polyhedron.

- *Symbolic concretization.* The symbolic concretization of a polyhedron, $\hat{\gamma}(P_1)$, is the conjunction of constraints in the constraint representation of P_1.

Recently, a new approach to implementing the polyhedral abstract domain was proposed [101]. This approach is based on linear programming; it attempts to improve the scalability of the analysis by eliminating conversions between the two representations. However, the abstract transformers are weakened as the result.

Chapter 3

Summarizing abstractions

Often, in program analysis, the situations arise in which universal properties of an unbounded collection of objects need to be established. The word “unbounded” means that the number of objects that must be taken into consideration by the analysis, may not be determined statically. That is, the number of objects may change from program run to program run depending on the user input or the run-time environment. For instance, if the program uses dynamic memory allocation, then the analysis may have to model an unbounded number of linked-list elements. If the program creates threads dynamically, then the analysis may need to track an unbounded number of instances of thread-local variables. Recursive-function invocations may create an unbounded number of instances of local variables on the run-time stack.

As we showed in Chapter 2, traditional numeric abstractions are only able to keep track of a fixed, finite set of objects. Thus, they cannot be used directly in the above situations. Maintaining a separate abstract-domain element for each possible number of objects (i.e., one-dimensional domain element to represent program states with one object, a two-dimensional domain element to represent states with two objects, etc.) is also infeasible because the object count is unbounded. For example, consider representing a linked-list of arbitrary length, each element of which stores the value 1. While the property is trivial numerically, it is impossible to represent it with the use of standard numeric abstractions: the analysis will need separate abstract-domain elements to represent lists of length 1, 2, 3, ..., etc.

The typical approach to reasoning about an unbounded number of objects (or simply a very large number of objects) is by employing abstraction. The set of objects is divided into a fixed number of groups based on certain criteria. Each group is then represented (we will say "*summarized*") by a single abstract object. The groups themselves need not be of bounded size. As an example, TVLA, a framework for shape analysis, uses *canonical abstraction* to create bounded-size representations of memory states [78, 100].

In this chapter, we present the techniques for using traditional abstract domains in this "summarizing" setting. The set of abstract objects will serve as the set of variables for the abstract domain, and the standard semantics of the domain will be lifted to account for *summary* abstract objects, i.e., those abstract objects that represent (summarize) more than one concrete object. To lift the semantics, we propose a non-standard concretization for the abstract-domain elements. Then, we show how to construct sound abstract transformers for the new concretization from existing abstract transformers.

3.1 Preliminaries

We need to extend some of the definitions of Chapter 2 and define some new notation to be able to talk about program states with a varying number of variables. In this section, we slightly extend the concrete semantics of the program and define the notion of *summarization*.

3.1.1 Extended Concrete Semantics

In contrast to Chapter 2, this chapter does not define a unified set of variables that is shared by all program states. Instead, each program state S is associated with a corresponding set of objects[1] U_S that exist in that state. The set U_S is referred to as the *universe* of S. A concrete program state maps each object in its universe to the corresponding value, i.e., $S : U_S \to \mathbb{V}$.

Because there is no fixed universe, we introduce a level of indirection to specify arbitrary expressions and conditionals in a state-independent manner: rather than referencing concrete objects directly, expressions and conditionals use free variables that are mapped to the concrete objects in

[1] In this section, we use the term "object" instead of the term "variable" to refer to the entities that the program manipulates. The "objects" in this chapter are more general than the "variables" in Chapter 2: they may refer to array elements, linked-list elements, etc. In §3.6, we will extend the abstraction to handle multiple numeric values attached to the same object.

a particular state by a state-specific function σ_S : *Fv* $\rightarrow U_S$, where *Fv* is a set of free variables. An arbitrary expression $\phi(w_1, \ldots, w_k)$, where $w_i \in$ *Fv*, is evaluated in the state S as follows (note that we use the non-deterministic expression semantics defined in §2.1.2):

$$[\![\phi(w_1, \ldots, w_k)]\!](S) = [\![\phi(w_1, \ldots, w_k)]\!]_{ND}(S(\sigma_S(w_1)), \ldots, S(\sigma_S(w_k)))$$

Similarly, an arbitrary boolean condition $\psi(w_1, \ldots, w_k)$, where $w_i \in$ *Fv*, is evaluated as follows:

$$[\![\psi(w_1, \ldots, w_k)]\!](S) = [\![\psi(w_1, \ldots, w_k)]\!]_{ND}(S(\sigma_S(w_1)), \ldots, S(\sigma_S(w_k)))$$

The semantics for the assign and assume transitions, as well as the collecting semantics of the program, are defined similarly to their counterparts in Chapter 2, with the exception that the above semantics for evaluating expressions and conditionals is used in place of the one in $\S 2.1.3$.

Let us stress one more time the primary difference between the concrete semantics defined in Chapter 2 and the concrete semantics defined above. In Chapter 2, *all* program states share the same universe and the mapping of variables in expressions and conditionals to the objects in the universe is program-state independent. This allows the concrete semantics of Chapter 2 to be effectively abstracted with the use of standard numeric abstract domains. In contrast, in this chapter, the universe varies from program state to program state; program states with different universes may arise at the same program location. Thus, existing numeric abstractions cannot be used directly.

3.1.2 Summarization

We assume that some abstraction is used to partition the universe of each concrete state S into m groups and that each group is represented by an object from a set $U^\sharp = \{u_1^\sharp, ..., u_m^\sharp\}$. We refer to $U^\sharp$ as the *abstract universe*. Let $\pi_S : U_S \rightarrow U^\sharp$ denote a function that maps concrete objects in state S to the corresponding abstract objects in $U^\sharp$. For convenience, we also define the inverse of this functions, $\pi_S^{-1} : U^\sharp \rightarrow \wp(U_S)$. Note that this scheme is general enough to specify a wide range of possible abstractions: the abstraction can be as simple as grouping together the objects created at the same allocation site, or it can be as complex as canonical abstraction, in which the grouping of concrete objects change from state to state. We refer to the abstract objects that represent more than a single concrete object as *summary* objects.

In this chapter, to simplify the presentation, we assume that there is only a single abstract universe $U^\sharp$. In practice, however, a number of separate abstract universes may be maintained by the analysis. For instance, in TVLA, each class of isomorphic shape graphs forms a separate abstract universe. The techniques in this chapter can still be applied to each individual abstract universe. However, the numeric abstractions that are associated with separate abstract universes must be collected and propagated separately.

Another simplification that we make is that, in each expression or conditional in the program, free variables are mapped to the same abstract objects uniformly across all of the program states that arise at that point. That is, we assume that there is a *uniform assignment function*[2] $\sigma^\sharp$ that agrees with $\pi_S \circ \sigma_S$ in each concrete program state S that arises at a given program point. This is often the case in practice because concrete objects are typically summarized based on the *memory-access patterns* that occur in the program's expressions, e.g., a C expression "p->v" may refer to different memory locations in different program states; however, all of these memory locations are likely to be represented by the same abstract object, e.g., the same node in the shape graph. If this is not the case, the situation can be handled by case splitting: because $U^\sharp$ is finite, there are only finitely many possible assignment functions that may arise.

3.1.3 Standard Abstract Domains Revisited

In Chapter 2, we showed that standard abstract domains provide *abstract transformers* $[\![\bar{x} \leftarrow \bar{\phi}]\!]^\sharp$ and $[\![\mathit{assume}(\psi)]\!]^\sharp$ that approximate the state transitions for sets of program states with a fixed universe (these were defined in §2.1.4). In this chapter, we will use these state transitions to transform sets of functions with fixed domain.[3] We slightly extend the notation, and refer to these state transitions as

$$[\![\bar{x} \leftarrow \bar{\phi}]\!]^{W,\sigma} \qquad \text{and} \qquad [\![\mathit{assume}(\psi)]\!]^{W,\sigma},$$

[2] In this context, "*uniform*" should be understood as uniform across *all* concrete states that arise at the program point where the corresponding expression or conditional is evaluated, but *not* as uniform for *all* program points.

[3] In Chapter 2, sets of program states are defined as sets of functions with a fixed domain *Vars*.

where W denotes the fixed domain for the functions, and σ maps free variables in the vector of expressions $\bar{\phi}$ and the conditional ψ to the corresponding members of W.

Recall from §2.1.4, that these transformers operate on each function in the set independently. That is, let $S : W \to \mathbb{V}$:

- The transformer $[\![\bar{x} \leftarrow \bar{\phi}]\!]^{W,\sigma}(S)$, for each function $f \in S$, evaluates the set of expressions $\bar{\phi}$ by substituting a value $S(\sigma(w_i))$ for each free variable w_i in $\bar{\phi}$, updates the mappings for objects $\sigma(\bar{x}[\cdot])$, and places the resulting function (or a collection of functions, if $\bar{\phi}$ is nondeterministic) into the output set. We define the function $\mathit{Values}_{\phi,\sigma} : (W \to \mathbb{V}) \to \wp(\mathbb{V})$, for an arbitrary k-ary $\phi \in \Phi$, as follows (let $f : W \to \mathbb{V}$ be an arbitrary function):

$$\mathit{Values}_{\phi,\sigma}(f) \quad \triangleq \quad [\![\phi[i]]\!]_{ND}(f(\sigma(w_1)), \ldots, f(\sigma(w_k))).$$

  Note that this function simply evaluates the expression ϕ in function f using the free-variable assignment function σ. Let $\bar{x} \in \mathit{Fv}^r$, $\bar{\phi} \in \Phi^r$, and let $i \in [1, r]$:

$$[\![\bar{x} \leftarrow \bar{\phi}]\!]^{W,\sigma}(S) = \left\{ f' \;\mid\; \exists f \in S \;\; \forall v \in W \;\; \begin{bmatrix} f'(v) \in \mathit{Values}_{\bar{\phi}[i],\sigma}(f) & \text{if } v = \sigma(x[i]) \\ f'(v) = f(v) & \text{otherwise} \end{bmatrix} \right\}$$

- The transformer $[\![\mathit{assume}(\psi)]\!]^{W,\sigma}(S)$, for each function $f \in S$, evaluates the condition ψ by substituting value $S(\sigma(w_i))$ for each free variable w_i in ψ. If the resulting value (set of values, if ψ is nondeterministic) equals *true* (contains *true*), the function f is added to the output set. Similarly to the assignment transformer, we define the function $\mathit{Values}_{\psi,\sigma} : (W \to \mathbb{V}) \to \wp(\mathbb{V})$, for an arbitrary k-ary $\psi \in \Psi$, as follows (let $f : W \to \mathbb{V}$ be an arbitrary function):

$$\mathit{Values}_{\psi,\sigma}(f) \quad \triangleq \quad [\![\psi]\!]_{ND}(f(\sigma(w_1)), \ldots, f(\sigma(w_k))).$$

  The actual transformer is defined as follows:

$$[\![\mathit{assume}(\psi)]\!]^{W,\sigma}(S) = \{f \in S \mid \mathit{true} \in \mathit{Values}_{\psi,\sigma}(f)\}$$

In the later sections of this chapter, we will use these transformers to manipulate sets of functions with a fixed domain $U^\sharp$, while using the uniform assignment function $\sigma^\sharp$ for binding free variables in expressions and conditionals to the objects in $U^\sharp$. According to the notation we use, the corresponding version of the transformers defined above will be denoted by

$$[\![\bar{x} \leftarrow \bar{\phi}]\!]^{U^\sharp,\sigma^\sharp} \qquad \text{and} \qquad [\![\mathit{assume}(\psi)]\!]^{U^\sharp,\sigma^\sharp}.$$

3.2 Summarizing abstraction

Let $Q \in \wp(\Sigma)$ denote a set of concrete program states. Let $U^\sharp$ be the set $\{u_1^\sharp, ..., u_m^\sharp\}$, and let the family of functions π_S, for $S \in Q$, specify how the universe of each state in Q is summarized. In this section, we show how to approximate the set of concrete program states Q by an element of an m-dimensional standard abstract domain.

Example 3.1 We will use the following example throughout this section to illustrate the abstraction. Let the set of concrete states be

$$Q = \left\{ \begin{array}{c} [u_1 \mapsto 1, u_2 \mapsto 2, u_3 \mapsto 3, u_4 \mapsto 4], \\ [u_5 \mapsto 2, u_6 \mapsto 3, u_7 \mapsto 4, u_8 \mapsto 5] \end{array} \right\}.$$

Let the abstract universe be $U^\sharp = \{u_1^\sharp, u_2^\sharp\}$, and let the concrete objects in both states be summarized as follows:

$$\begin{aligned} \pi_1 &= \left[u_1 \mapsto u_1^\sharp, \quad u_2 \mapsto u_2^\sharp,\ u_3 \mapsto u_2^\sharp,\ u_4 \mapsto u_2^\sharp \right], \\ \pi_2 &= \left[u_5 \mapsto u_1^\sharp, \quad u_6 \mapsto u_2^\sharp,\ u_7 \mapsto u_2^\sharp,\ u_8 \mapsto u_2^\sharp \right]. \end{aligned}$$

The abstraction proceeds in three conceptual steps (expressed as Galois insertions: $\langle \alpha_1, \gamma_1 \rangle$, $\langle \alpha_2, \gamma_2 \rangle$, and $\langle \alpha_3, \gamma_3 \rangle$):

1. *Abstracting function domains.* In the first step, we abstract each function $S : U_S \to \mathbb{V}$ in Q by a function $S_1 : U^\sharp \to \wp(\mathbb{V})$: each abstract object $u^\sharp \in U^\sharp$ is mapped to the set that contains S-images of objects represented by $u^\sharp$:

$$Q_1 = \alpha_1(Q) = \left\{ S_1 : U^\sharp \to \wp(\mathbb{V}) \;\middle|\; \begin{array}{c} \forall u^\sharp \in U^\sharp \ \left[S_1(u^\sharp) = \left\{ S(u) \mid u \in \pi_S^{-1}(u^\sharp) \right\} \right] \\ \text{for some } S \in Q \end{array} \right\}$$

The respective concretization function is defined as follows:

$$Q' = \gamma_1(Q_1) = \left\{ S : U_S \to \mathbb{V} \;\middle|\; \begin{array}{c} \forall u^\sharp \in U^\sharp \;\; \forall a \in S_1(u^\sharp) \;\; \exists u \in \pi_S^{-1}(u^\sharp) \;\; [\, S(u) = a \,] \\ \text{for some } S_1 \in Q_1 \end{array} \right\}$$

Example 3.2 The application of this abstraction step to the set of states in the Ex. 3.1 yields the following set:

$$Q_1 = \alpha_1(Q) = \left\{ \left[u_1^\sharp \mapsto \{1\},\; u_2^\sharp \mapsto \{2,3,4\} \right],\; \left[u_1^\sharp \mapsto \{2\},\; u_2^\sharp \mapsto \{3,4,5\} \right] \right\}$$

Note that this abstraction step loses relationships among values associated with concrete objects that are summarized together. For instance, the program state

$$S = [u_1 \mapsto 1, u_2 \mapsto 4, u_3 \mapsto 3, u_4 \mapsto 2]$$

is also in $\gamma_1(Q_1)$.

2. *Flattening function images.* In the second step, each function $S_1 : U^\sharp \to \wp(\mathbb{V})$ in the set Q_1 is represented by a set of functions with signature $U^\sharp \to \mathbb{V}$ by exhaustively enumerating for each $u^\sharp \in U^\sharp$ all possible mappings from the set $S_1(u^\sharp)$:

$$Q_2 = \alpha_2(Q_1) = \left\{ S_2 : U^\sharp \to \mathbb{V} \;\middle|\; \begin{array}{c} \forall u^\sharp \in U^\sharp \;\; [\, S_2(u^\sharp) \in S_1(u^\sharp) \,] \\ \text{for some } S_1 \in Q_1 \end{array} \right\}$$

The respective concretization function is defined as follows:

$$Q_1' = \gamma_2(Q_2) = \left\{ S_1 : U^\sharp \to \wp(\mathbb{V}) \;\mid\; \alpha_2(S_1) \subseteq Q_2 \right\}$$

Example 3.3 Applying this abstraction step to the set of functions Q_1 from Ex. 3.2 yields:

$$Q_2 = \alpha_2(Q_1) = \left\{ \begin{array}{c} \left\{ u_1^\sharp \mapsto 1, u_2^\sharp \mapsto 2 \right\}, \left\{ u_1^\sharp \mapsto 1, u_2^\sharp \mapsto 3 \right\}, \left\{ u_1^\sharp \mapsto 1, u_2^\sharp \mapsto 4 \right\}, \\ \left\{ u_1^\sharp \mapsto 2, u_2^\sharp \mapsto 3 \right\}, \left\{ u_1^\sharp \mapsto 2, u_2^\sharp \mapsto 4 \right\}, \left\{ u_1^\sharp \mapsto 2, u_2^\sharp \mapsto 5 \right\} \end{array} \right\}$$

This abstraction steps loses some of the relationships among the values of concrete objects that are *not* summarized together. For instance, the program state

$$S = [u_1 \mapsto 1, u_2 \mapsto 2, u_3 \mapsto 2, u_4 \mapsto 2]$$

is in the concretization $(\gamma_1 \circ \gamma_2)(Q_2)$, but not in the concretization $\gamma_1(Q_1)$. However, some relationships are still preserved: for instance, one property that is preserved is that, if the value of the concrete object summarized by $u_1^\sharp$ is 1, then the values of the concrete objects summarized by $u_2^\sharp$ must be in the set $\{2, 3, 4\}$.

3. *Standard numeric abstraction.* In the last step, we use standard abstraction techniques from §2.5 to represent the set of functions $Q_2 \in \wp(U^\sharp \to \mathbb{V})$ with an element of an abstract numeric domain.

Example 3.4 In polyhedral domain, the set Q_2 from Ex. 3.3 is represented by the following polyhedron:

$$Q_3 = \left\{ 1 \le u_1^\sharp \le 2,\ u_1^\sharp + 1 \le u_2^\sharp \le u_1^\sharp + 3 \right\}.$$

The first two steps form the crux of the abstraction. Throughout the rest of this chapter we primarily focus on the partial abstraction $\alpha_2 \circ \alpha_1$, rather than on the full abstraction all the way to the abstract domain. We will use a special notation, a superscript $\flat$, to distinguish sets and set operations at this level from their counterparts at the concrete level, e.g., let $\Sigma^\flat = \wp(U^\sharp \to \mathbb{V})$ denote the set of all *partial* abstract states.

Trivially, $\bot^\flat = \emptyset^\flat$ exactly represents the empty set of concrete states. Similarly, $\top^\flat = \Sigma^\flat$ represents all possible concrete states. To see this, observe that $\gamma_2(\top^\flat) = \wp(U^\sharp \to \wp(\mathbb{V}))$, from which γ_1 can construct any concrete state in Σ. Next, observe that $\cup^\flat$ and $\cap^\flat$ are sound abstractions of $\cup$ and $\cap$, respectively. This follows trivially from the construction of the abstraction. Furthermore, note that $\cap^\flat$ is exact; that is,

$$(\gamma_1 \circ \gamma_2)(S_1^\flat \cap^\flat S_2^\flat) = (\gamma_1 \circ \gamma_2)(S_1^\flat)\ \cap\ (\gamma_1 \circ \gamma_2)(S_2^\flat) \quad \text{for all } S_1^\flat, S_2^\flat \in \Sigma^\flat,$$

while $\cup^\flat$ overapproximates $\cup$:

$$(\gamma_1 \circ \gamma_2)(S_1^\flat \cup^\flat S_2^\flat) \supseteq (\gamma_1 \circ \gamma_2)(S_1^\flat)\ \cup\ (\gamma_1 \circ \gamma_2)(S_2^\flat) \quad \text{for all } S_1^\flat, S_2^\flat \in \Sigma^\flat,$$

Thus, at the level of the abstract domain, the operations $\sqcup$ and $\sqcap$, as well as $\top$ and $\bot$, are sound with respect to the concrete semantics, because they soundly approximate $\cup^\flat$, $\cap^\flat$, $\top^\flat$, and $\bot^\flat$ (see Chapter 2). Also, ∇ and Δ form valid widening and narrowing operators for the overall abstraction.

In the next few sections, we will concentrate on defining sound abstract transformers for the assignment transition and the assume transition. The first two abstraction steps approximate a set of functions with arbitrary domains by a set of functions with a fixed domain $U^\sharp$. Thus, it is tempting to use transformers from §3.1.3, e.g.,

$$[\![\bar{x} \leftarrow \bar{\phi}]\!]^\flat = [\![\bar{x} \leftarrow \bar{\phi}]\!]^{U^\sharp,\sigma^\sharp} \qquad \text{and} \qquad [\![\mathit{assume}(\psi)]\!]^\flat = [\![\mathit{assume}(\psi)]\!]^{U^\sharp,\sigma^\sharp} \tag{3.1}$$

where $\sigma^\sharp$ is the uniform assignment function (see §3.1.2). The transformers $[\![\cdot]\!]^\flat$ could then trivially be approximated by the corresponding transformers $[\![\cdot]\!]^\sharp$ of the underlying abstract domain. However, as the following example demonstrates, doing so is *not sound*.

In Ex. 3.5, we show that the abstract transformers defined in Eqn. (3.1) are unsound by picking a concrete state from the concretization of a partial abstract state $S_1^\flat$, transforming that concrete state according to the concrete semantics, and showing that the resulting state is not in the concretization of $S_2^\flat$, a partial abstract state obtained by applying the transformer from Eqn. (3.1) to $S_1^\flat$.

Example 3.5 Let $U^\sharp = \{u_1^\sharp, u_2^\sharp\}$, where $u_2^\sharp$ is summary; let $S_1^\flat$ be the partial abstract state:

$$S_1^\flat = \left\{ \left\{ u_1^\sharp \mapsto 1, u_2^\sharp \mapsto 2 \right\}, \left\{ u_1^\sharp \mapsto 1, u_2^\sharp \mapsto 3 \right\}, \left\{ u_1^\sharp \mapsto 1, u_2^\sharp \mapsto 4 \right\} \right\}.$$

Let $S_2^\flat$ denote the result of transforming the abstract state $S_1^\flat$ with respect to the assignment $v_1 \leftarrow v_2$, where variable v_1 is bound to $u_1^\sharp$, and variable v_2 is bound to $u_2^\sharp$, i.e.,

$$S_2^\flat = [\![v_1 \leftarrow v_2]\!]^\flat(S_1^\flat) = [\![v_1 \leftarrow v_2]\!]^{U^\sharp,\left[v_1 \mapsto u_1^\sharp, v_2 \mapsto u_2^\sharp\right]}(S_1^\flat).$$

The transformer $[\![\bar{x} \leftarrow \bar{\phi}]\!]^{U^\sharp,\sigma^\sharp}$ updates each function in $S_1^\flat$ independently and yields:

$$S_2^\flat = \left\{ \left\{ u_1^\sharp \mapsto 2, u_2^\sharp \mapsto 2 \right\}, \left\{ u_1^\sharp \mapsto 3, u_2^\sharp \mapsto 3 \right\}, \left\{ u_1^\sharp \mapsto 4, u_2^\sharp \mapsto 4 \right\} \right\}.$$

Now, consider the concrete state $S_1 = [u_1 \mapsto 1, u_2 \mapsto 2, u_3 \mapsto 3, u_4 \mapsto 4]$, in which the concrete objects are summarized as follows:

$$\pi_{S_1} = \left[u_1 \mapsto u_1^\sharp,\ u_2 \mapsto u_2^\sharp, u_3 \mapsto u_2^\sharp, u_4 \mapsto u_2^\sharp \right]$$

Note that S_1 is clearly represented by $S_1^\flat$, i.e., $S_1 \in (\gamma_1 \circ \gamma_2)(S_1^\flat)$. Now, let S_2 denote the result of transforming concrete state S_1 with respect to the assignment $v_1 \leftarrow v_2$. Let us assume that the free variables are assigned as follows: $\sigma_{S_1} = [v_1 \mapsto u_1, v_2 \mapsto u_2]$, so that v_1 is bound to the (single) concrete object represented by $u_1^\sharp$, and v_2 is bound to one of the concrete objects represented by $u_2^\sharp$. As a result, $S_2 = [u_1 \mapsto 2, u_2 \mapsto 2, u_3 \mapsto 3, u_4 \mapsto 4]$.

Now, if the abstract transformer $[\![v_1 \leftarrow v_2]\!]^\flat$ is sound, then S_2 must be represented by $S_2^\flat$, i.e., $S_2 \in (\gamma_1 \circ \gamma_2)(S_2^\flat)$. However, this is not the case. To see this, observe that the abstraction of S_2,

$$(\alpha_2 \circ \alpha_1)(S_2) = \left\{ \left\{ u_1^\sharp \mapsto 2, u_2^\sharp \mapsto 2 \right\}, \left\{ u_1^\sharp \mapsto 2, u_2^\sharp \mapsto 3 \right\}, \left\{ u_1^\sharp \mapsto 2, u_2^\sharp \mapsto 4 \right\} \right\}$$

is not contained in $S_2^\flat$, i.e., $(\alpha_2 \circ \alpha_1)(S_2) \not\sqsubseteq^\flat S_2^\flat$. Thus, the abstract transformers defined in Eqn. (3.1) are indeed not sound.

Intuitively, the abstract transformers in Eqn. (3.1) fail because they transform each function in a partial abstract state $S^\flat$ independently. However, there are certain dependences between the functions in $S^\flat$ that must be respected. In particular, each concrete state in $(\gamma_1 \circ \gamma_2)(S^\flat)$ is represented by multiple functions in $S^\flat$. Also, each function in S can be used to (partially) represent multiple concrete states in $(\gamma_1 \circ \gamma_2)(S^\flat)$. In the next section, we define several operations that will be useful in the construction of sound version of transformers for $[\![\bar{x} \leftarrow \bar{\phi}]\!]^\flat$ and $[\![\textit{assume}(\psi)]\!]^\flat$.

3.3 New operations

In this section, we define four operations on sets of functions with fixed domains: two primary operations, *add* and *drop*, and two derived operations, *expand* and *fold*. To be able to lift an existing abstract domain for use in a summarizing abstraction, the existing domain will have to provide sound approximations for these operations. The derived operations can be expressed in terms of the primary operations *add* and *drop*, plus existing operations for transforming sets of functions (i.e., $[\![\bar{x} \leftarrow \bar{\phi}]\!]^{W,\sigma}$ and $[\![\textit{assume}(\psi)]\!]^{W,\sigma}$). However, having explicit definitions for the derived operations greatly simplifies the presentation. Also, from a practical standpoint, for many domains it is possible to implement the derived operations much more efficiently, compared to direct use of the general definitions we supply in this section. We will use the following notation throughout the section: U will denote a fixed set of objects, S will denote a set of functions with domain U: $S \in \wp(U \to \mathbb{V})$.

3.3.1 The *add* operation.

Intuitively, the $\mathit{add}_u(S)$ operation injects a new object u into the universe U (we assume that initially $u \notin U$). No restrictions are imposed on the value associated with u. In the concrete semantics, the $\mathit{add}_u(S)$ operation models the allocation of a new object u.

$$[\![\mathit{add}_u]\!](S) = \{ f' : U \cup \{u\} \to \mathbb{V} \;\mid\; f'(u) \in \mathbb{V} \quad \text{and} \quad \exists f \in S \;\text{ s.t. }\; \forall t \in U \;\; [\, f'(t) = f(t)\,] \} \,.$$

In a standard abstract domain $\mathbb{D}$, the operation add_u corresponds to adding a new dimension to the multidimensional space and embedding the original subset into that space. Many existing implementations of abstract domains already support this operation. We think it is reasonable to expect that the abstract transformer $[\![\mathit{add}_u]\!]^\sharp$ can be implemented *exactly* in any abstract domain.

3.3.2 The *drop* operation.

The $\mathit{drop}_u(S)$ removes object u from the universe U (we assume that initially $u \in U$). Each function in S is transformed to "forget" the mapping for u. In the concrete semantics, drop_u models the deallocation of object u:

$$[\![\mathit{drop}_u]\!](S) = \{ f' : U \setminus \{u\} \to \mathbb{V} \;\mid\; \exists f \in S \;\text{ s.t. }\; \forall t \in U \setminus \{u\} \;\; [\, f'(t) = f(t)\,] \}$$

In a standard abstract domain $\mathbb{D}$, the operation drop_u corresponds to existentially projecting out the dimension associated with the object u. The majority of standard abstract domain implementations already provide this operation. In the polyhedral abstract domain, the drop_u operation can be computed precisely. We expect that this holds for the majority of existing abstract domains.

3.3.3 The *fold* operation

The $\mathit{fold}_{u,v}$ operation formalizes the concept of *summarizing* objects together: the objects u and v are summarized together; u denotes the resulting summary object; and v is dropped from the universe. In particular, $\mathit{fold}_{u,v}$ transforms each function $f \in S$ into two functions f' and f'' over domain $U \setminus \{v\}$, such that $f'(u) = f(u)$ and $f''(u) = f(v)$:[4]

$$\textit{fold}_{u,v}(S) = \left\{ f', f'' : U \setminus \{v\} \to \mathbb{V} \;\middle|\; \exists f \in S \;\left[\begin{array}{c} \forall t \in U \setminus \{u,v\} \;\; [f'(t) = f''(t) = f(t)] \\ \wedge\, f'(u) = f(u) \;\wedge\; f''(u) = f(v) \end{array} \right] \right\}$$

The $\textit{fold}_{u,v}$ operation can be expressed in terms of the standard transformers for sets of functions and the *drop* operation as follows:

$$[\![\textit{fold}_{u,v}]\!](S) = [\![\textit{drop}_v]\!](S \;\cup\; [\![w_1 \leftarrow w_2]\!]^{U,\sigma}(S)),$$

where $\sigma = [w_1 \mapsto u, w_2 \mapsto v]$.

In a standard abstract domain $\mathbb{D}$, the abstract transformer $[\![\textit{fold}_{u,v}]\!]^\sharp$ can be trivially implemented by directly using the approximations of the operations in the above definition:

$$[\![\textit{fold}_{u,v}]\!]^\sharp(S^\sharp) = [\![\textit{drop}_v]\!]^\sharp(S^\sharp \;\sqcup\; [\![u \leftarrow v]\!]^\sharp(S^\sharp)), \tag{3.2}$$

However, in some domains (notably, in weakly-relational and non-relational domains), internal representation details can be used to implement this operation much more efficiently, than a direct implementation of Eqn. (3.2), which involves duplicating the abstract domain element $S^\sharp$.

Note that implementations of $\textit{fold}_{u,v}$ operation are likely to lose some precision. This is due to the $\sqcup$ operator in the definition of $[\![\textit{fold}_{u,v}]\!]^\sharp$: in numeric abstractions, join operators tend to incur precision loss.

3.3.4 The *expand* operation

In contrast to the *fold* operation, the *expand* operation formalizes a partial concretization process. Semantically, $\textit{expand}_{u,v}$ *materializes* an object v from the group represented by summary object u. Because the summarizing abstraction loses distinctions among objects that are summarized together, the most that we can infer about the value of v is that it must have the same relationship to the values of other objects as the value of u, but there should be no relationships

[4] Note that this corresponds to the first two steps of the summarizing abstraction: the first abstraction step transforms each function $f \in S$, into function f_1, such that $f_1(u) = \{f(u), f(v)\}$; the second abstraction step breaks the function f_1 into the functions f' and f''.

between the values of v and u. We assume that $v \notin U$.

$$\textit{expand}_{u,v}(S) = \left\{ f : U \cup \{v\} \to \mathbb{V} \;\middle|\; \exists f_1, f_2 \in S \left[\begin{array}{c} \forall t \in U \setminus \{u\}\ [f(t) = f_1(t) = f_2(t)] \\ \wedge\, f(u) = f_1(u) \ \wedge\ f(v) = f_2(u) \end{array} \right] \right\}$$

The $\textit{expand}_{u,v}$ operation can be expressed in terms of the standard transformers for sets of functions and the *add* operation as follows:

$$[\![\textit{expand}_{u,v}]\!](S) = [\![\textit{add}_v]\!](S) \ \cap\ [\![\langle w_1, w_2\rangle \leftarrow \langle w_2, w_1\rangle]\!]^{U \cup \{v\},\sigma}([\![\textit{add}_v]\!](S)),$$

where $\sigma = [w_1 \mapsto u, w_2 \mapsto v]$. Note that we add an unconstrained object v to the domain of S, make a copy of the resulting set, *swap* objects v and u in one of the sets, and take the intersection of the two sets. Intuitively, one of the two sets creates a proper mapping for $f(u)$, the other set (in which u and v are swapped) creates a proper mapping for $f(v)$. The intersection makes sure that the functions in the resulting set have proper mappings for both $f(u)$ and $f(v)$.

Similarly to the *fold* operation, the *expand* operation can be trivially implemented in an abstract domain $\mathbb{D}$ as follows:

$$[\![\textit{expand}_{u,v}]\!]^\sharp(S^\sharp) = [\![\textit{add}_v]\!]^\sharp(S^\sharp) \ \ \sqcap\ \ [\![\langle u, v\rangle \leftarrow \langle v, u\rangle]\!]^\sharp \circ [\![\textit{add}_v]\!]^\sharp(S^\sharp)$$

However, given knowledge of the internal representation used by the domain, a more efficient implementation can usually be constructed. For instance, in the domain of polyhedra, the operation $[\![\textit{expand}_{u,v}]\!]^\sharp$ can be implemented by rewriting the constraint system: for each constraint in which object u participates, a similar constraint with u substituted by v is added to the constraint system.

Finally, in most numeric domains, both the meet operation and $[\![\textit{add}_v]\!]^\sharp$ are exact. Thus, if the domain is able to perform the assignment that swaps u and v exactly, then the above implementation for the $[\![\textit{expand}_{u,v}]\!]^\sharp$ is also exact.

3.4 Abstract Semantics

In this section, with the help of the new operations from §3.3, we define sound transformers for $[\![x \leftarrow \phi]\!]^\flat$ and $[\![\textit{assume}(\psi)]\!]^\flat$. In the case of $[\![x \leftarrow \phi]\!]^\flat$, to simplify the presentation, we only consider assignment transitions that update a single variable. The extension to *parallel assignments*, which update multiple variables simultaneously, is trivial (see §3.4.1).

Let $S^\flat \in \wp(U^\sharp \to \mathbb{V})$ be a partial abstract state. Ideally, we would like to define abstract transformers that operate on each function $f \in S^\flat$ independently. However, as we pointed out before, the functions in $S^\flat$ are not independent. In particular, an arbitrary function $f \in S^\flat$ may belong to the representations of multiple concrete states in the concretization of $S^\flat$. The transformation of each concrete state may affect the function f in a different way. We will deal with this issue by *expanding* certain objects in $U^\sharp$ based on the particular expression (or condition) that appears in the state transition. Intuitively, we will *compile* certain knowledge about the expression (or condition) into the functions in $S^\flat$ to make the resulting functions independent with respect to that expression (or condition). Then, we will use the standard transformer for sets of functions (which relies on the independence of the functions in the set) to perform the actual transformation. Finally, we will eliminate the extra objects, which we introduced into $U^\sharp$.

In contrast to the unsound semantics defined in Eqn. (3.1), which attempted to use the transformers for sets of functions with fixed domain directly, namely,

$$\llbracket \bar{x} \leftarrow \bar{\phi} \rrbracket^\flat = \llbracket \bar{x} \leftarrow \bar{\phi} \rrbracket^{U^\sharp,\sigma^\sharp} \qquad \text{and} \qquad \llbracket \mathit{assume}(\psi) \rrbracket^\flat = \llbracket \mathit{assume}(\psi) \rrbracket^{U^\sharp,\sigma^\sharp}$$

the sound transformers have the following form:

$$\llbracket x \leftarrow \phi \rrbracket^\flat = \llbracket \mathit{drop}_\phi \rrbracket \circ \llbracket x \leftarrow \phi \rrbracket^{U^\sharp_\phi,\sigma^\sharp_\phi \cup [x \mapsto \sigma^\sharp(x)]} \circ \llbracket \mathit{expand}_\phi \rrbracket,$$

and

$$\llbracket \mathit{assume}(\psi) \rrbracket^\flat = \llbracket \mathit{drop}_\psi \rrbracket \circ \llbracket \mathit{assume}(\psi) \rrbracket^{U^\sharp_\psi,\sigma^\sharp_\psi} \circ \llbracket \mathit{expand}_\psi \rrbracket,$$

where expand_ϕ, drop_ϕ, expand_ψ, and drop_ψ are the extensions of operations *expand* and *drop*, and $U^\sharp_\phi$, $\sigma^\sharp_\phi$, $U^\sharp_\psi$, and $\sigma^\sharp_\psi$ are the extensions of the abstract universe $U^\sharp$ and the uniform assignment function $\sigma^\sharp$. These extensions are specific to the expression ϕ and the condition ψ. In the rest of this section, we detail the construction of these transformers and justify their soundness.

3.4.1 Assignment transitions: $x \leftarrow \phi(w_1, \ldots, w_k)$

Let $x \leftarrow \phi(w_1, \ldots, w_k)$ be an arbitrary assignment transition, and let $\sigma^\sharp$ be the uniform assignment function that maps w_i to the objects in $U^\sharp$. To transform a function $f \in S^\flat$, we need to be able to compute the set of values to which the expression ϕ evaluates in each of the concrete states in the concretization of $S^\flat$ that have f in its representation. We define the function

$$\textit{Values}_{S^\flat,\phi} : (U^\sharp \to \mathbb{V}) \to \wp(\mathbb{V}),$$

which maps each $f \in S^\flat$ to the corresponding set of values. The function is defined as follows:

$$\textit{Values}_{S^\flat,\phi}(f) = \left\{ [\![\phi(w_1, \ldots, w_k)]\!](S) \;\mid\; S \in (\gamma_1 \circ \gamma_2)(S^\flat) \;\wedge\; f \in (\alpha_2 \circ \alpha_1)(S) \right\}. \tag{3.3}$$

Note that if the functions in $S^\flat$ were independent (that is, if there was a one-to-one correspondence between the concrete states in $(\gamma_1 \circ \gamma_2)(S^\flat)$ and functions in $S^\flat$), the above set could be evaluated as follows:

$$\textit{Values}_{S^\flat,\phi}(f) = [\![\phi(w_1, \ldots, w_k)]\!]_{ND}(f(\sigma^\sharp(w_1)), \ldots, f(\sigma^\sharp(w_k))).$$

However, we do not have this luxury.

A variable w_i in the expression ϕ is either mapped to a summary abstract object or to a non-summary abstract object. If $\sigma^\sharp(w_i)$ is a non-summary object, the situation is simple: each $f \in S^\flat$ maps each non-summary abstract object directly to the value of the corresponding concrete object.[5] So, when evaluating the expression, we can substitute the value $f(\sigma^\sharp(w_i))$ for the variable w_i.

However, if w_i is mapped to a summary object, the situation is more complicated: due to the second abstraction step, the values of the concrete objects represented by $\sigma^\sharp(w_i)$ may have been spread over multiple functions in $S^\flat$. Thus, substituting $f(\sigma^\sharp(w_i))$ for the variable w_i only accounts for a single possible value and is, consequently, unsound. We handle this case by *expanding* the abstract object $\sigma^\sharp(w_i)$ (i.e., by applying the $\textit{expand}_{\sigma^\sharp(w_i),v^\sharp}$ operation), and substituting the value $f(v^\sharp)$ for the variable w_i when evaluating the expression.

Intuitively, if some variable w_i is mapped to a summary abstract object $u^\sharp$ by the assignment function $\sigma^\sharp$, we *materialize* an object $v^\sharp$ from the collection of objects represented by $u^\sharp$, and

[5] This follows from the construction of the abstraction: the first abstraction step causes non-summary abstract objects to be mapped to singleton sets of values.

change the assignment function to map w_i to $v^\sharp$. At a lower level, the *expand*$_{u^\sharp,v^\sharp}$ operation transforms each function $f \in S^\flat$ into a set of functions $\{f'\}$, such that each of the f' agree with f on all $u_i^\sharp \in U^\sharp$, but map the new object $v^\sharp$ to each value that the concrete objects represented by $u^\sharp$ may have in concrete states whose abstractions contain function f.

Evaluation of Numeric Expressions. Let $\phi(w_1, \ldots, w_k) \in \Phi$ be an arbitrary k-ary expression. We assume, without loss of generality, that $\sigma^\sharp$ maps the first $\hat{k}$ variables of ϕ to summary objects, i.e., $1 \le \hat{k} \le k$. We define several pieces of notation to simplify the definition of the transformers. First, we define a ϕ-specific *expand* operation, which expands all of the summary abstract objects to which the variables w_i are mapped by $\sigma^\sharp$:

$$[\![\mathit{expand}_\phi]\!] = [\![\mathit{expand}_{\sigma^\sharp(w_{\hat{k}}),u^\sharp_{m+\hat{k}}}]\!] \circ \ldots \circ [\![\mathit{expand}_{\sigma^\sharp(w_1),u^\sharp_{m+1}}]\!].$$

We denote the expanded abstract universe by $U^\sharp_\phi = U^\sharp \cup \left\{u^\sharp_{m+1}, \ldots, u^\sharp_{m+\hat{k}}\right\}$. Additionally, we define a ϕ-specific mapping of free variables to objects in $U^\sharp_\phi$ as follows:

$$\sigma^\sharp_\phi(w_i) = \begin{cases} u^\sharp_{m+i} & \text{if } \sigma^\sharp(w_i) \text{ is summary, i.e., if } i \le \hat{k} \\ \sigma^\sharp(w_i) & \text{otherwise} \end{cases}$$

Finally, we define a clean-up operation that eliminates all of the extra objects created by *expand*$_\phi$:

$$[\![\mathit{drop}_\phi]\!] = [\![\mathit{drop}_{u^\sharp_{m+1}}]\!] \circ \ldots \circ [\![\mathit{drop}_{u^\sharp_{m+\hat{k}}}]\!]$$

Armed with these primitives, we define an approximation for the function *Values*$_{S^\flat,\phi}$ as follows:

$$[\![\mathit{Values}_{S^\flat,\phi}]\!]^\flat(f) = \left\{ [\![\phi]\!]_{ND}(f'(\sigma^\sharp_\phi(w_1)), \ldots, f'(\sigma^\sharp_\phi(w_k))) \;\middle|\; \begin{array}{c} f' \in [\![\mathit{expand}_\phi]\!](S^\flat) \wedge \\ \forall u \in U^\sharp \, [f(u) = f'(u)] \end{array} \right\} \quad (3.4)$$

Note that, compared to the definition of the function *Values*$_{S^\flat,\phi}$ in Eqn. (3.3), which involves enumerating the concrete states represented by $S^\flat$, the definition of $[\![\mathit{Values}_{S^\flat,\phi}]\!]^\flat$ is much more operational: the operation $[\![\mathit{expand}_\phi]\!]$ (which can be trivially approximated by the underlying abstract domain) is applied to the original set of functions; the expression can then be evaluated *independently* for each function in the expanded set. The following lemma states that the definition in Eqn. (3.4) expresses the set *Values*$_{S^\flat,\phi}(f)$, for any $f \in S^\flat$, exactly.

Lemma 3.6 Let $S^\flat \in \wp(U^\sharp \to \mathbb{V})$ be a partial abstract state. And let $\phi \in \Phi$ be an arbitrary expression. Then,

$$\forall f \in S^\flat \ \left[\mathit{Values}_{S^\flat,\phi}(f) = [\![\mathit{Values}_{S^\flat,\phi}]\!]^\flat(f)\right]$$

Proof. *See the Appendix.* ■

Let us now define the abstract transformer for the assignment transition. Two cases must be considered: (i) the case when variable x is mapped to a *non-summary* object, and (ii) the case when variable x is mapped to a *summary* object. We start with the simpler case of $\sigma^\sharp(x)$ being a *non-summary* object $u^\sharp$.

Non-summary assignment. The non-summary abstract object $u^\sharp = \sigma^\sharp(x)$ represents a single concrete object, the one that is being assigned to. Thus, the transformer may directly update the mapping for $u^\sharp$ in each function $f \in S^\flat$ to reflect the effect of the assignment. The values to which the object $u^\sharp$ must be mapped by the function $f \in S^\flat$ come from the set $\mathit{Values}_{S^\flat,\phi}(f)$ (recall, that the set $\mathit{Values}_{S^\flat,\phi}(f)$ contains values to which the expression ϕ evaluates in *every* concrete state S represented by $S^\flat$, such that S also contains the function f in its abstraction; thus, as each such concrete state S changes as the result of the assignment, the function f must change accordingly). That is:

$$[\![x \leftarrow \phi]\!]^\flat(S^\flat) = \left\{ f' : U^\sharp \to \mathbb{V} \ \middle|\ \exists f \in S^\flat \ \left[\begin{array}{c} u^\sharp = \sigma^\sharp(x) \ \wedge \ f'(u^\sharp) \in \mathit{Values}_{S^\flat,\phi}(f) \\ \wedge \ \forall t \in U^\sharp \setminus \{u^\sharp\} \ [f'(t) = f(t)] \end{array} \right] \right\} \quad (3.5)$$

Note however, that this definition is not operational: that is, it is not clear how to apply it to an element of an abstract domain.

We would like to express the definition in Eqn. (3.5) in terms of a standard assignment transformer for sets of functions with a fixed domain and the operations for the evaluation of numeric expressions, which we have defined above. These operations can be approximated trivially by the corresponding operations of the abstract domain. We rely on the idea from the definition of $[\![\mathit{Values}_{S^\flat,\phi}]\!]^\flat$. First, we apply the operation $[\![\mathit{expand}_\phi]\!]$ to the partial abstract state $S^\flat$: as a result, for each function $f \in S^\flat$ and each value $a \in \mathit{Values}_{S^\flat,\phi}$, the expanded set contains a function f', such that (i) f' agrees with f on all objects in $U^\sharp$; and (ii) the expression ϕ, when evaluated on f'

with variable-assignment function $\sigma^\sharp_\phi$, yields the value a. We apply the standard assignment transformer from §3.1.3 to the expanded set of functions to update the mapping for the object $\sigma^\sharp(x)$ in each f' to a corresponding value from the set $\mathit{Values}_{S^\flat,\phi}(f)$. Finally, we get rid of the extra objects introduced by the $[\![\mathit{expand}_\phi]\!]$ operation by applying the operation $[\![\mathit{drop}_\phi]\!]$. The abstract transformer for the assignment transition is expressed as follows:

$$[\![x \leftarrow \phi]\!]^\flat(S^\flat) = [\![\mathit{drop}_\phi]\!] \circ [\![x \leftarrow \phi]\!]^{U^\sharp_\phi, \sigma^\sharp_\phi \cup [x \mapsto \sigma^\sharp(x)]} \circ [\![\mathit{expand}_\phi]\!](S^\flat)$$

Note that the assignment function $\sigma^\sharp_\phi$ is extended with the mapping for the left-hand-side variable x. Also, note that because, according to Lem. 3.6, $[\![\mathit{Values}_{S^\flat,\phi}]\!]^\flat = \mathit{Values}_{S^\flat,\phi}$, this definition is equivalent to the definition in Eqn. (3.5).

Theorem 3.7 The abstract transformer $[\![x \leftarrow \phi]\!]^\flat$ is *sound*. That is, for an arbitrary partial abstract state $S^\flat$,

$$(\gamma_1 \circ \gamma_2)([\![x \leftarrow \phi]\!]^\flat(S^\flat)) \supseteq [\![x \leftarrow \phi]\!]((\gamma_1 \circ \gamma_2)(S^\flat))$$

Proof. *See the Appendix.* ■

At the level of the abstract domain, the transformer is implemented by using the approximations for the above operations:

$$[\![x \leftarrow \phi]\!]^\sharp_\star(S^\sharp) = [\![\mathit{drop}_\phi]\!]^\sharp \circ [\![\sigma(x)^\sharp \leftarrow \phi^\sharp]\!]^\sharp \circ [\![\mathit{expand}_\phi]\!]^\sharp(S^\sharp),$$

where $\phi^\sharp$ is obtained by substituting each variable w_i in ϕ with $\sigma^\sharp_\phi(w_i)$. Note that because the abstract domain operations only approximate the operations in the definition of the abstract transformer, the precision of the transformer implementation depends heavily on how precise these approximations are.

Summary assignment. In case $\sigma^\sharp(x)$ is a summary object, the situation is more complicated: the object $\sigma^\sharp(x)$ represents not only the concrete object that is updated by the assignment, but also other concrete objects whose values do not change. As a result, the transformer must perform a weak update. One obvious way of implementing a weak update is to duplicate the abstract state, update one of the copies of the state, and join both copies together. However, the availability of the *fold* operation allows us to to perform such weak updates *in-place*, i.e., we use a new object $u^\sharp$ to collect the values that are assigned to $\sigma^\sharp(x)$ (the object $\sigma^\sharp(x)$ is left unmodified), and then fold the object $u^\sharp$ into $\sigma^\sharp(x)$:

$$[\![x \leftarrow \phi]\!]^\flat_{weak}(S^\flat) = [\![\mathit{drop}_\phi]\!] \circ [\![\mathit{fold}_{\sigma^\sharp(x),u^\sharp}]\!] \circ [\![x \leftarrow \phi]\!]^{U^\sharp_\phi \cup \{u^\sharp\}, \sigma^\sharp_\phi \cup [x \mapsto u^\sharp]} \circ [\![\mathit{add}_{u^\sharp}]\!] \circ [\![\mathit{expand}_\phi]\!](S^\flat)$$

It can be easily seen that the above transformer is sound: it relies on the non-summary transformer, which we have shown to be sound (see Thm. 3.7). Then, it weakens the result of the sound transformer further by joining it with the values that the target object had before the transformation.

At the level of the abstract domain, the transformer is implemented by using the approximations for the above operations:

$$[\![x \leftarrow \phi]\!]^\sharp_{\star\star}(S^\sharp) = [\![\mathit{drop}_\phi]\!]^\sharp \circ [\![\mathit{fold}_{\sigma^\sharp(x),u^\sharp}]\!]^\sharp \circ [\![u^\sharp \leftarrow \phi^\sharp]\!]^\sharp \circ [\![\mathit{add}_{u^\sharp}]\!]^\sharp \circ [\![\mathit{expand}_\phi]\!]^\sharp(S^\sharp),$$

where $\phi^\sharp$ is obtained by substituting each variable w_i in ϕ with $\sigma^\sharp_\phi(w_i)$.

Parallel assignments. The task of extending the above transformers to handle parallel assignment $\bar{x} \leftarrow \bar{\phi}$ is trivial, but tedious. Thus, we will omit the derivation of the transformer, and just point out a few things that should be taken into consideration. Let us assume the assignment updates r variables in parallel, i.e., $\bar{x} \in \mathit{Fv}^r$ and $\bar{\phi} \in \Phi^r$:

- The operation $\mathit{expand}_{\bar{\phi}}$ and $\mathit{drop}_{\bar{\phi}}$ must be defined to operate on the entire vector $\bar{\phi}$.
- The set $\mathit{Values}_{S^\flat,\bar{\phi}}(f)$ for some $f \in S^\flat$ must be defined to collect r-tuples of values in $\mathbb{V}$, i.e., $\mathit{Values}_{S^\flat,\bar{\phi}}(f) \subseteq \mathbb{V}^r$.
- Finally, some variables in $\bar{x}$ may map to summary objects, while some others may map to non-summary objects. The corresponding *add* and *fold* operations must be added for the components of $\bar{x}$ that are mapped to summary abstract objects by $\sigma^\sharp$.

3.4.2 Assume transitions: *assume*$(\psi(w_1, \ldots, w_k))$

This section recreates the constructions of §3.4.1, but for the purpose of evaluating conditional expressions. Because the material is very similar to §3.4.1, we keep the discussion to a minimum.

Let *assume*(ψ) denote an arbitrary assume transition, where $\psi \in \Psi$. Also, let $\sigma^\sharp$ denote the function that binds free variables in ψ to the corresponding abstract objects in $U^\sharp$. Let $S^\natural$ be a partial abstract state. Consider a function $f \in S^\natural$: this function may belong to the representation of multiple concrete states in the concretization of $S^\natural$. If at least one of these concrete states satisfies the conditional expression ψ, the function f must be added to the resulting partial abstract state (because f is a part of the abstraction of that concrete state).

Much like in the case of the assignment transition, we define the function $\mathit{Values}_{S^\natural,\psi}$, which maps each function $f \in S^\natural$ to the set of values to which ψ evaluates in each concrete state S represented by $S^\natural$, such that f belongs to the representation of S:

$$\mathit{Values}_{S^\natural,\psi}(f) = \left\{ [\![\psi(w_1,\ldots,w_k)]\!](S) \ \mid \ S \in (\gamma_1 \circ \gamma_2)(S^\natural) \ \wedge \ f \in (\alpha_2 \circ \alpha_1)(S) \right\}. \tag{3.6}$$

The only difference with the assignment transition is that the set $\mathit{Values}_{S^\natural,\psi}(f)$ may contain at most two values: *true* and *false*.

The abstract assume transformer for the partial abstraction is defined with the use of function $\mathit{Values}_{S^\natural,\psi}$ as follows:

$$[\![\mathit{assume}(\psi)]\!]^\natural(S^\natural) = \left\{ f \in S^\natural \ \mid \ \mathit{true} \in \mathit{Values}_{S^\natural,\psi}(f) \right\}. \tag{3.7}$$

However, this definition is not directly computable, because it relies on the enumerating a possibly infinite set of concrete states. To construct an operational definition, we need to have a better way to construct the set $\mathit{Values}_{S^\natural,\psi}$ for the functions in $S^\natural$.

Evaluation of Conditional Expressions. Let $\psi(w_1,\ldots,w_k) \in \Psi$ be an arbitrary k-ary conditional expression. We assume, without loss of generality, that $\sigma^\sharp$ maps the first $\hat{k}$ variables of ϕ to summary objects, i.e., $1 \le \hat{k} \le k$. We define new notation, which is similar in spirit to the notation defined for the numeric expressions.

First, we define a ψ-specific *expand* operation, which expands all of the summary abstract objects to which the variables w_i are mapped by $\sigma^\sharp$:

$$[\![\mathit{expand}_\psi]\!] = [\![\mathit{expand}_{\sigma^\sharp(w_{\hat{k}}),u^\sharp_{m+\hat{k}}}]\!] \circ \ldots \circ [\![\mathit{expand}_{\sigma^\sharp(w_1),u^\sharp_{m+1}}]\!].$$

We denote the expanded abstract universe by $U^\sharp_\psi = U^\sharp \cup \left\{u^\sharp_{m+1}, ..., u^\sharp_{m+\hat{k}}\right\}$. Additionally, we define a ψ-specific mapping of free variables to objects in $U^\sharp_\psi$ as follows:

$$\sigma^\sharp_\psi(w_i) = \begin{cases} u^\sharp_{m+i} & \text{if } \sigma^\sharp(w_i) \text{ is summary, i.e., if } i \leq \hat{k} \\ \sigma^\sharp(w_i) & \text{otherwise} \end{cases}$$

Finally, we define a clean-up operation that eliminates all of the extra objects created by *expand*$_\psi$:

$$[\![\mathit{drop}_\psi]\!] = [\![\mathit{drop}_{u^\sharp_{m+1}}]\!] \circ ... \circ [\![\mathit{drop}_{u^\sharp_{m+\hat{k}}}]\!]$$

Similarly to §3.4.1, we define an approximation for the function *Values*$_{S^\flat,\psi}$ as follows:

$$[\![\mathit{Values}_{S^\flat,\psi}]\!]^\flat(f) = \left\{ [\![\phi]\!]_{ND}(f'(\sigma^\sharp_\psi(w_1)), \ldots, f'(\sigma^\sharp_\psi(w_k))) \;\middle|\; \begin{array}{c} f' \in [\![\mathit{expand}_\psi]\!](S^\flat) \wedge \\ \forall u \in U^\sharp \,[f(u) = f'(u)] \end{array} \right\}$$

Lemma 3.8 Let $S^\flat \in \wp(U^\sharp \to \mathbb{V})$ be a partial abstract state. And let $\psi \in \Psi$ be an arbitrary expression. Then,

$$\forall f \in S^\flat \;\left[\mathit{Values}_{S^\flat,\psi}(f) = [\![\mathit{Values}_{S^\flat,\psi}]\!]^\flat(f)\right]$$

Proof. *See the Appendix.* ■

Assume transformer. Lem. 3.8 gives us an effective way to compute the sets *Values*$_{S^\flat,\psi}(f)$ for the functions $f \in S^\flat$. We apply the operation $[\![\mathit{expand}_\psi]\!](S^\flat)$ to the set $S^\flat$. Note that if for some function $f \in S^\flat$ the set *Values*$_{S^\flat,\psi}(f)$ contains true, then there must be at least one function f' in $[\![\mathit{expand}_\psi]\!](S^\flat)$, such that (i) f' is constructed from f (i.e., f' agrees with f on all objects in $U^\sharp$), and (ii) the conditional expression ψ evaluates to *true* on f' under the variable-assignment function $\sigma^\sharp_\psi$. We collect all such f' by filtering the expanded set with the standard assume transformer (using variable-assignment function $\sigma^\sharp_\psi$). In the last step, we eliminate the extra objects introduced by the $[\![\mathit{expand}_\psi]\!]$ operation: thus, all the f' that remain in the set after filtering are reduced to the corresponding functions f from $S^\flat$ (this follows from (i) above). The abstract transformer for the assume transition is expressed as follows:

$$[\![\mathit{assume}(\psi)]\!]^\flat(S^\flat) = [\![\mathit{drop}_\psi]\!] \circ [\![\mathit{assume}(\psi)]\!]^{U^\sharp_\psi,\sigma^\sharp_\psi} \circ [\![\mathit{expand}_\psi]\!](S^\flat).$$

$u_1^\sharp$	$u_2^\sharp$
1	2
1	3
1	4

(a) $S_1^\flat$

$u_1^\sharp$	$u_2^\sharp$	$u_3^\sharp$
1	2	2
1	2	3
1	2	4
1	3	2
1	3	3
1	3	4
1	4	2
1	4	3
1	4	4

(b) $S_{exp}^\flat$

$u_1^\sharp$	$u_2^\sharp$	$u_3^\sharp$
2	2	2
3	2	3
4	2	4
2	3	2
3	3	3
4	3	4
2	4	2
3	4	3
4	4	4

(c) $S_{asgn}^\flat$

$u_1^\sharp$	$u_2^\sharp$
2	2
3	2
4	2
2	3
3	3
4	3
2	4
3	4
4	4

(d) $S_2^\flat$

Figure 3.1 The application of a partial summarizing abstract transformer: (a) $S_1^\flat$ is the initial partial abstract state; (b) $S_{exp}^\flat$ is obtained from $S_1^\flat$ by expanding summary object $u_2^\sharp$; (c) $S_{asgn}^\flat$ is obtained from $S_{exp}^\flat$ by preforming an assignment $u_1^\sharp \leftarrow u_3^\sharp$; (d) the resulting state $S_2^\flat$ is obtained from $S_{asgn}^\flat$ by dropping the object $u_3^\sharp$.

Theorem 3.9 The abstract transformer $[\![\mathit{assume}(\psi)]\!]^\flat$ is *sound*. That is, for an arbitrary partial abstract state $S^\flat$,

$$(\gamma_1 \circ \gamma_2)([\![\mathit{assume}(\psi)]\!]^\flat(S^\flat)) \supseteq [\![\mathit{assume}(\psi)]\!]((\gamma_1 \circ \gamma_2)(S^\flat))$$

Proof. *See the Appendix.* ■

At the level of the abstract domain, the transformer is implemented by using the approximations for the above operations:

$$[\![\mathit{assume}(\psi)]\!]_\star^\sharp(S^\sharp) = [\![\mathit{drop}_\psi]\!]^\sharp \circ [\![\mathit{assume}(\psi^\sharp)]\!]^\sharp \circ [\![\mathit{expand}_\psi]\!]^\sharp(S^\sharp),$$

where $\psi^\sharp$ is obtained by substituting each variable w_i in ψ by $\sigma_\psi^\sharp(w_i)$.

3.4.3 Example

Let us illustrate the abstract transformers defined above by revisiting Ex. 3.5. Recall the example setting: $U^\sharp = \{u_1^\sharp, u_2^\sharp\}$, where $u_2^\sharp$ is summary, and $S_1^\flat$ is the partial abstract state:

$$S_1^\flat = \left\{ \left\{ u_1^\sharp \mapsto 1, u_2^\sharp \mapsto 2 \right\}, \left\{ u_1^\sharp \mapsto 1, u_2^\sharp \mapsto 3 \right\}, \left\{ u_1^\sharp \mapsto 1, u_2^\sharp \mapsto 4 \right\} \right\}.$$

We would like to transform $S_1^\flat$ with respect to the assignment $v_1 \leftarrow v_2$, where the free variables v_1 and v_2 are mapped to the objects in $U^\sharp$ as follows: $\sigma^\sharp = \left[v_1 \mapsto u_1^\sharp, v_2 \mapsto u_2^\sharp \right]$.

Fig. 3.1 shows the intermediate sets of functions that are computed by the summarizing (partial) abstract state transformer $[\![v_1 \leftarrow v_2]\!]^\flat$. We represent sets of functions with some fixed domain U graphically, as tables: each row corresponds to an object in U, and each column represents a single function in the set. Fig. 3.1(a) shows this representation for the set $S_1^\flat$.

The variable v_2 is mapped by $\sigma^\sharp$ to a summary object $u_2^\sharp$; thus, first, the transformer *expands* the object $u_2^\sharp$, yielding the set $S_{exp}^\flat$ (see Fig. 3.1(b)):

$$S_{exp}^\flat = [\![\textit{expand}_{u_2^\sharp, u_3^\sharp}]\!](S_1^\flat)$$

Note that, as the result of the *expand*, each function in $S_1^\flat$ is transformed into three functions in $S_{exp}^\flat$, one for each possible value that a concrete object represented by $u_2^\sharp$ may hold: the object $u_3^\sharp$ is mapped to the corresponding value by each function.

Next, the standard assignment transformer for functions with fixed domain $U^\sharp \cup \{u_3^\sharp\}$ is applied to the set $S_{exp}^\flat$ (note that the variable v_2 is mapped to the object $u_3^\sharp$, which was created by the *expand* operation):

$$S_{asgn}^\flat = [\![v_1 \leftarrow v_2]\!]^{U^\sharp \cup \{u_3^\sharp\}, \left[v_1 \mapsto u_1^\sharp, v_2 \mapsto u_3^\sharp\right]}(S_{exp}^\flat)$$

The resulting set of functions $S_{asgn}^\flat$ is shown in Fig. 3.1(c). The effect of the transformation is readily seen in the table form: in each row of the table, the assignment moves the value from the third column ($u_3^\sharp$) to the first column ($u_1^\sharp$).

Finally, the resulting partial abstract state $S_2^\flat$ is obtained by dropping the object $u_3^\sharp$ from $S_{asgn}^\flat$, which corresponds to eliminating the column corresponding to $u_3^\sharp$ from the table:

$$S_2^\flat = [\![\textit{drop}_{u_3^\sharp}]\!](S_{asgn}^\flat).$$

The final result is shown in Fig. 3.1(d). Note that $S_2^\natural$ represents exactly the set of concrete states that arises as a result of applying the corresponding assignment transition to the set of concrete states represented by $S_1^\natural$.

3.5 Symbolic Concretization

A symbolic-concretization function $\hat{\gamma}$ for a given abstract domain expresses the meaning of an abstract element of that domain as a formula in some logic.[6] Symbolic concretization is a useful device for formalizing the set of properties that can be represented with a given abstraction. Also, symbolic concretization can be used in conjunction with theorem provers to automatically construct *best* abstract transformers for certain domains [96, 118]. For most existing numeric domains, with the exception of arithmetic automata [12, 13], a symbolic-concretization function $\hat{\gamma}$ can be constructed trivially. For instance, an element of a polyhedral abstract domain can be expressed as a conjunction of the constraints that form the polyhedron.

In this section, we discuss the symbolic-concretization function $\hat{\gamma}_\star$ for the summarizing abstraction. Because the summarizing abstraction utilizes a standard abstract domain in its last abstraction step, we will express $\hat{\gamma}_\star$ in terms of $\hat{\gamma}$ for that abstract domain.

First, lets formally define the symbolic concretization for standard numeric domains. Let *Vars* $= \{u_1, \ldots, u_n\}$ be a set of variables. Thus, each program state is a function with signature *Vars* $\rightarrow \mathbb{V}$. Let $\mathbb{D}^n$ denote an abstract domain that is able to represent sets of such program states. Let $S^\sharp$ denote an element of $\mathbb{D}^n$. Then, $\hat{\gamma}(S^\sharp)$ yields a logical formula $\psi(v_1, \ldots, v_n)$, such that for all program states S : *Vars* $\rightarrow \mathbb{V}$ the following holds:

$$\psi(S(u_1), \ldots, S(u_n)) = \begin{cases} \textit{true} & \text{if } S \in \gamma(S^\sharp) \\ \textit{false} & \text{if } S \notin \gamma(S^\sharp) \end{cases}$$

Example 3.10 Recall the polyhedron Q_3 from Ex. 3.4.

$$\hat{\gamma}(Q_3) \quad \triangleq \quad (1 \le v_1 \le 2) \ \wedge \ (v_1 + 1 \le v_2 \le v_1 + 3)$$

[6] For instance, for standalone numeric domains, Presburger arithmetic provides a sufficient language for expressing the meaning of domain elements. However, if numeric abstraction is a part of a more complex abstraction, a more expressive logic may be required. For instance, the meaning of a TVLA structure with associated numeric state can only be expressed in first-order logic.

In the summarizing abstraction setting, the universe differs from program state to program state. However, for each program state S we have a mapping $\pi_S : U_S \to U^\sharp$, where $U^\sharp = \{u_1^\sharp, \ldots, u_k^\sharp\}$. In the course of the abstraction, sets of program states are represented by sets of functions with the signature $U^\sharp \to \mathbb{V}$, which are in turn abstracted by elements of some abstract domain $\mathbb{D}^k$. Let $S^\sharp$ denote an element of $\mathbb{D}^k$, and let $\gamma_\star = \gamma_1 \circ \gamma_2 \circ \gamma_3$ denote the concretization function for the summarizing abstraction. The set of concrete program states represented by $S^\sharp$ can be logically characterized as follows:

$$\forall u_1 \in \pi_S^{-1}(u_1^\sharp) \ldots \forall u_k \in \pi_S^{-1}(u_k^\sharp) \left[\, \hat{\gamma}(S^\sharp)(S(u_1), \ldots, S(u_k)) \,\right] = \begin{cases} \textit{true} & \text{if } S \in \gamma_\star(S^\sharp) \\ \textit{false} & \text{if } S \notin \gamma_\star(S^\sharp) \end{cases}$$

Note that we cannot express $\hat{\gamma}_\star$ fully in the sense that we cannot express the limited quantification $\forall u_i \in \pi_S^{-1}(u_i^\sharp)$ without knowing how the universe of each program state is partitioned. Rather, these parts of formula will have to be filled in by the "client" abstraction, such as TVLA.

Example 3.11 Recall the polyhedron Q_3 from Ex. 3.4.

$$\hat{\gamma}_\star(Q_3) \quad \triangleq \quad \forall u_1 \in \pi_S^{-1}(u_1^\sharp)\ \forall u_2 \in \pi_S^{-1}(u_2^\sharp)\ [\, 1 \le u_1 \le 2\ \wedge\ u_1 + 1 \le u_2 \le u_1 + 3\,]$$

From this construction we learn that the summarizing abstraction is able to represent certain universal properties of the concrete objects that are summarized together. The class of properties that can be represented is limited by the capabilities of the standard abstract domain that is used by the last abstraction step.

3.6 Support for Multiple Values

In some circumstances, it may be desirable to associate multiple quantities with an individual concrete object. For instance, concrete objects may represent objects of a C++ class that defines multiple numeric fields; they may represents threads that have multiple thread-local variables; or they may represent activation records that have multiple local variables associated with them. In Chapter 4, we will need to summarize sets of array elements, which will have two quantities associated with them: the value and the index. In this section, we show how to extend the abstraction with the ability to handle multiple values.

In the following, we assume that each concrete object has q fields associated with it. We will denote the set of fields by $F = \{f_1, \ldots, f_q\}$, and we will use the dot notation $u.f$ to refer to a field f associated with object u. Also, we assume that the free variables in the expressions and conditionals are bound to objects rather than fields, i.e., the expressions will look as follows: $v_1.f_1 + v_2.f_4$.

The straightforward way of handling this situation is to *explode* the universe of concrete objects by including a separate object for each field of each original concrete object. The abstract universe must be exploded similarly, and π_S must be updated to summarize the fields properly, i.e.,

$$\hat{U}_S = \left\{ u_i.f_j \;\middle|\; \begin{array}{c} u_i \in U_S, \\ 1 \leq j \leq q \end{array} \right\} \quad \hat{U^\sharp} = \left\{ u_i^\sharp.f_j \;\middle|\; \begin{array}{c} u_i^\sharp \in U^\sharp, \\ 1 \leq j \leq q \end{array} \right\} \quad \hat{\pi}_S(u_i.f_j) = \pi_S(u_i).f_j$$

Then, the summarizing abstraction can be applied exactly as described in §3.2. The disadvantage of this approach is that the fields of the concrete objects, which are summarized together, are represented by separate summary abstract objects. Thus, certain relationships among fields that are associated with the same concrete object may not be represented. For instance, consider array elements that have two fields, *value* and *index*, associated with them. The following property cannot be represented with this approach:

$$\forall u \in \pi^{-1}(A^\sharp)\ \left[\, u.\textit{value} = 2 \times u.\textit{index} + 3 \,\right],$$

where $A^\sharp$ represents all of the concrete array elements. The reason for this becomes apparent when we recall the symbolic concretization function $\hat{\gamma}_\star$ from §3.5: with this approach, the only quantification choices that we have are $\forall u.\textit{value} \in \pi^{-1}(A^\sharp.\textit{value})$ and $\forall u.\textit{index} \in \pi^{-1}(A^\sharp.\textit{index})$, but not $\forall u \in \pi^{-1}(A^\sharp)$.

An alternative approach is to allow concrete objects to be associated with tuples of values. That is, each program state S now becomes a function $S : U_S \to \mathbb{V}^q$, i.e., each concrete object u is now mapped to a vector that contains values of its fields, e.g., the value of $u.f_i$ is given by $S(u)[i]$. The summarizing abstraction can be applied to sets of such states only partially: the application of the first two abstraction steps yields a set of functions with signature $U^\sharp \to \mathbb{V}^q$, which, however, cannot be directly represented by an element of a standard abstract domain.

To handle this issue we add an extra abstraction step, which represents sets of functions with signature $U^\sharp \rightarrow \mathbb{V}^q$ by sets of functions with signature $\hat{U}^\sharp \rightarrow \mathbb{V}$, where $\hat{U}^\sharp$ is the abstract universe that is exploded in the same way as in the straightforward approach, i.e., $\hat{U}^\sharp = \left\{ u_i^\sharp.f_j \mid u_i^\sharp \in U^\sharp,\, j = 1..q \right\}$. The resulting set of functions can be trivially approximated with an element of a $(k \times q)$-dimensional numeric abstract domain.

For the abstract transformers from §3.4 to work on this representation, we must provide the operations $\mathit{add}_{u^\sharp}$, $\mathit{drop}_{u^\sharp}$, $\mathit{expand}_{u^\sharp,v^\sharp}$, and $\mathit{fold}_{u^\sharp,v^\sharp}$ that can operate on sets of functions in $\wp(\hat{U}^\sharp \rightarrow \mathbb{V})$. In particular, these operations must now operate *simultaneously* on groups of objects in $\hat{U}^\sharp$. The semantics for the operations $\mathit{add}_{u^\sharp}$ and $\mathit{drop}_{u^\sharp}$ can be defined as compositions of *add* and *drop* operations for the objects in $\hat{U}^\sharp$ that represent individual fields of $u^\sharp$, that is

$$[\![\mathit{add}_{u^\sharp}]\!] = [\![\mathit{add}_{u^\sharp.f_1}]\!] \circ \ldots \circ [\![\mathit{add}_{u^\sharp.f_q}]\!] \qquad \text{and} \qquad [\![\mathit{drop}_{u^\sharp}]\!] = [\![\mathit{drop}_{u^\sharp.f_1}]\!] \circ \ldots \circ [\![\mathit{drop}_{u^\sharp.f_q}]\!].$$

The $\mathit{expand}_{u^\sharp,v^\sharp}$ and $\mathit{fold}_{u^\sharp,v^\sharp}$ cannot be defined as the composition of *expand* and *fold* operations for the objects that represent individual fields. These operations must truly operate in parallel on all of the fields of $u^\sharp$ and $v^\sharp$. We define them as follows (let $S \in \wp(\hat{U}^\sharp \rightarrow \mathbb{V})$):

$$[\![\mathit{fold}_{u^\sharp,v^\sharp}]\!](S) = [\![\mathit{drop}_{v^\sharp}]\!](S \;\cup\; [\![\bar{x} \leftarrow \bar{y}]\!]^{\hat{U}^\sharp,\sigma}(S)),$$

where $\sigma = \left[x_1 \mapsto u^\sharp.f_1, \ldots, x_q \mapsto u^\sharp.f_q,\; y_1 \mapsto v^\sharp.f_1, \ldots, y_q \mapsto v^\sharp.f_q\right]$. And

$$[\![\mathit{expand}_{u^\sharp,v^\sharp}]\!](S) = [\![\mathit{add}_{v^\sharp}]\!](S) \;\cap\; [\![\langle \bar{x}, \bar{y}\rangle \leftarrow \langle \bar{y}, \bar{x}\rangle]\!]^{\hat{U}^\sharp \cup \{v^\sharp.f_1,\ldots,v^\sharp.f_q\},\sigma} \circ [\![\mathit{add}_{v^\sharp}]\!](S),$$

where $\sigma = \left[x_1 \mapsto u^\sharp.f_1, \ldots, x_q \mapsto u^\sharp.f_q,\; y_1 \mapsto v^\sharp.f_1, \ldots, y_q \mapsto v^\sharp.f_q\right]$.

3.7 Numeric extension of TVLA

In this section, we sketch how the techniques of this chapter were integrated into TVLA, a three-valued-logic-based framework for shape analysis [78, 100]. We extended the TVLA specification language with primitives for specifying numeric conditional expressions in logic formulae, and for specifying numeric updates in TVLA actions (i.e., specifications for program-state transitions). The TVLA implementation was extended to maintain and query the numeric state associated with each abstract memory configuration. We used the resulting numeric extension of TVLA to implement the prototype for the array analysis tool, which we describe in more detail in Chapter 4.

TVLA models concrete states by first-order logical structures: the universe of a logical structure represents the set of concrete objects; the properties of concrete objects are encoded with the help of a finite set of predicates. Historically, the objects in the universe of a structure are called *nodes*.[7]

Abstract states are represented by *three-valued logical structures*, which (conceptually) are constructed by applying canonical abstraction to the sets of concrete states. The abstraction is defined by a vector of unary predicates, and summarizes together the concrete objects whose predicates evaluate to the same vector of values. When nodes are summarized together, the interpretations of predicates for those nodes are joined using the information-order semi-lattice of 3-valued logic.[8]

TVLA distinguishes between two types of predicates: *core* predicates and *instrumentation* predicates. Core predicates are the predicates that are necessary to model the concrete states. Instrumentation predicates are defined in terms of core predicates, and are introduced to capture properties that are lost by the abstraction.

TVLA provides two operations to dynamically regroup summarized nodes:

- A *focus* operation replaces a three-valued structure by a set of more precise three-valued structures that represent the same set of concrete states as the original structure. Usually, focus is used to “materialize” a non-summary node from a summary node. (The structures resulting from a focus are not necessarily images of canonical abstraction, in the sense that they may have multiple nodes for which the abstraction predicates evaluate to the same values.)

- A *coerce* operation is a cleanup operation that “sharpens” updated three-valued structures by making them comply with a set of globally defined integrity constraints.

[7] In shape analysis, these objects correspond to the nodes of the shape graph.

[8] In the information order, $0 \sqsubseteq 1/2$, $1 \sqsubseteq 1/2$, and 0 and 1 are incomparable.

The numeric extension of TVLA allows to associate a number of numeric fields with each concrete object in the universe. When abstraction is applied, the values of numeric fields are approximated by an element of a summarizing abstract domain that is attached to each 3-valued structure.

To specify numeric properties, TVLA's specification language was extended to allow numeric comparisons to be used in logical formulas, e.g., $x(v) \wedge data(v) \geq 0$, where x is a unary predicate and $data$ is a numeric field. Numeric updates are specified via two kinds of formulas: (i) a numeric update formula, e.g., $data(v_1) = 2 \times data(v_2) + 3$, and (ii) a logical formula that binds free variables in the numeric formula to the nodes in the structure, e.g., $x(v_1) \wedge n(v_1, v_2)$. Both comparisons and updates are evaluated by determining the assignment of abstract objects to the free variables in the formula, and then executing a corresponding method of the summarizing abstract domain.

The focus and coerce operations are slightly harder to implement: to impose the numeric properties from focus formulas and from global integrity constraints onto the element of a summarizing domain, we use the assume operation provided by the domain. Node duplication, which is inherent to focus, is handled by the *expand* operation. Similarly, to reflect the merge of two nodes, the *fold* operation is used.

3.8 Related Work

Standard abstract domains. In Chapter 2, we discussed at length existing numeric abstract domains. These domains included *non-relational domains*: intervals [28], congruences [52], and interval congruences [82]; *weakly relational domains*: zones [84], octagons [85], zone congruences [86], TVPLI [109], and octahedra [23, 24]; as well as *relational domains*: linear equalities [68], linear congruences [52], polyhedra [32, 57], trapezoidal congruences [81], and arithmetic automata [12, 13]. All these domains make the assumption that there is a fixed, finite number of numeric objects that need to be tracked. In contrast, our work provides techniques for performing static analysis in the presence of an unbounded number of concrete numeric objects (which are then summarized by some fixed, finite number of *summary* objects).

TVLA. The closest approach to the summarizing abstraction is the framework for shape analysis based on 3-valued logic [78, 100]. In fact, this framework provided the primary motivation for our work. In contrast to summarizing abstraction, in 3-valued-logic abstraction only Boolean values (in the form of unary predicates) are associated with the concrete elements. Also, the abstraction for sets of Boolean values is hardcoded into the framework (a simple non-relational abstraction is used), whereas the summarizing abstraction allows any existing numeric domain to be used to represent the final abstract states.

Pointer and shape analysis. Many shape-analysis techniques use certain numeric information to enhance the expressibility of the shape abstraction. Typically, the numeric information represents things like the length of list segments, or the number of recursive "`p = p->next`" dereferences that need to be made to get to a corresponding memory location. Below, we give a brief survey of such techniques. In all of them, the number of numeric quantities that are tracked by the analysis is actually fixed and finite. Thus, standard numeric domains are employed to abstract that numeric information.

Yavuz-Kahveci and Bultan present an algorithm for shape analysis in which numeric information is attached to summary nodes [117]; the numeric quantity associated with the summary node $u^\sharp$ in a shape-graph S indicates the number of concrete nodes that are represented by $u^\sharp$. An element of some standard numeric abstract domain is attached to the shape graph to keep track of these quantities. This approach differs from our work in the following way: in [117], each summarized object contributes 1 to the quantity associated with the summary object; in contrast, in the summarizing abstraction, when objects are summarized together, the effect is to collect their values into a set.

Alain Deutsch proposed an alias analysis [35] that is based on representing access paths to particular objects as a regular expression annotated with numeric quantities, e.g., access path $p \rightarrow \texttt{next}^i \rightarrow \texttt{prev}^j$ specifies an object that is reachable from pointer p by i consecutive dereferences of the field `next`, followed by j consecutive dereferences of the field `prev`. To check whether two access paths specify the same object, certain numeric relationships between the numeric quantities that appear in both access paths are checked. An element of some standard numeric domain is used to track the above numeric quantities for each pair of access paths.

Arnaud Venet proposed a *non-uniform* pointer analysis [112, 113] (that is, the analysis is able to distinguish among elements of collections). The analysis is based on associating each dynamically-allocated object with a numeric timestamp, i.e., distinct objects have distinct timestamps. In the course of the analysis, the numeric relationships among the object's timestamp and its position in the collection (e.g., its array index) are captured. In the end, alias queries are reduced to checking certain numeric properties.

Chapter 4

Analysis of Array Operations

An array is a simple and efficient data structure that is heavily used in practice. In many cases, to verify the correctness of programs that use arrays, a program analyzer must be able to discover relationships among values of array elements, as well as their relationships to scalar variables in the program. However, the implementations of array operations are typically parameterized by scalar variables that bear certain numeric relationships to the actual size of the array. Thus, verification of a property in the presence of such operations usually requires establishing that the desired property holds for any possible values of the parameters with which the operation may be invoked. In other words, the analysis may have to reason about arrays of varying, possibly unbounded size.

Reasoning about unbounded collections of objects poses a challenge for existing numeric analysis techniques. However, the summarizing abstractions, which we introduced in Chapter 3, specifically target this situation. Thus, one possible approach to this problem is to *summarize* the potentially-unbounded set of array elements with a fixed number of abstract array elements (in the extreme—with a single summary element[1]), and use summarizing numeric abstractions to represent and manipulate the numeric state of the program. The potential problem with this approach is the precision loss due to *weak updates*: that is, updates that modify a single concrete object in a summarized collection must be modeled by accumulating into—rather than overwriting—the value kept for the summarized collection; because the summarizing abstraction can only maintain the universal properties of the entire summarized collection, the update to a single element may not be captured precisely (see the discussion of summary assignments in §3.4.1).

[1] This technique is called *array smashing* [14]

In this chapter, we develop a static-analysis framework for analyzing array operations. The framework is based on *canonical abstraction* [78, 100], a family of abstractions that employs a client-defined set of properties to partition, and summarize, a potentially-unbounded set of concrete objects, i.e., the objects that share similar properties are summarized together. In our framework, array elements are partitioned with respect to the numeric relationships between the indices of the array elements and the values of scalar variables that are used to index into the array. Note that the resulting abstraction is *storeless*; that is, there is no fixed connection between a concrete array element c and an abstract array element a that represents it: after the properties of concrete element c change (e.g., if the scalar variable that is used to partition the array is incremented), c may be represented by an abstract element other than a.

The analysis uses summarizing numeric abstractions from Chapter 3 to keep track of the values and indices of array elements. In our framework, indices of array elements are modeled explicitly; that is, two numeric quantities are associated with each array element: (i) the actual value of the element and (ii) its index.

Given an array and a set of scalar variables, we partition the elements of the array as follows. The array elements that are indexed by scalar variables are placed by themselves into individual groups, which are represented by non-summary abstract elements. Array segments in-between the indexed elements are also placed into individual groups, but these groups are represented by summary abstract elements. In practice, the *partitioning set* for an array will contain the scalar variables that are used *syntactically* to index into the array (i.e., variable `i` is added to the partitioning set of array `a`, if `a[i]` appears in the code of the program). Note that in this partitioning scheme, indexed array elements are always represented by non-summary abstract elements; thus, a *strong update* can always be applied when there is a write to such an element. Furthermore, for the array operations that scan the array linearly, this partitioning scheme separates the array elements that have already been processed (e.g., initialized or sorted), from the ones that have not; this allows the summarizing abstraction to capture stronger universal properties for the processed array elements.

```
void copy_array(int a[], int b[], int n) {
   i ← 0; (⋆)
   while(i < n) {
      b[i] ← a[i];
      i ← i + 1;
   } (⋆⋆)
}
```

Figure 4.1 Simple array-copy function.

One limitation of the summarizing abstraction is that it cannot capture the relations among individual array elements that are summarized together. For instance, the property that the elements of an array are in a certain order (e.g., the array is sorted) cannot be represented with a summarizing abstraction. To address this issue, in §4.3.3, we introduce auxiliary predicates that are attached to each abstract array element and capture numeric properties that are beyond the capabilities of a summarizing numeric domain on its own. At present, the auxiliary predicates, as well as their abstract transformers, are supplied manually. We expect this step to be automated in the future.

We implemented a prototype of the array analysis with the use of the numeric extension of the TVLA tool [78], which we described in §3.7. With this prototype implementation, we were able to analyze successfully several small, but challenging examples, including verifying the correctness of an insertion-sort implementation.

4.1 Overview of Array Analysis

In this section, we illustrate the analysis using a simple example. The procedure in Fig. 4.1 copies the contents of array a into array b. Both arrays are of size n, which is specified as a parameter to the procedure. Suppose that the analysis has already determined some facts about values stored in array a. For instance, suppose that the values of elements in a range from -5 to 5. At the exit of the procedure, we would like to establish that the values stored in array b also range from -5 to 5. Furthermore, we would like to establish this property for any reasonable array size, i.e., for all values of variable n greater than or equal to 1.

Our technique operates by partitioning the unbounded number of concrete array elements into a bounded number of groups. Each group is represented by an abstract array element. The partitioning is done based on numeric relationships between the indices of array elements and the value of loop induction variable `i`. In particular, for the example in Fig. 4.1, our technique will represent the elements of the two arrays with indices less than `i` by two *summary* abstract objects (which we will denote by $a_{<i}$ and $b_{<i}$, respectively). Array elements with indices greater than `i` are represented by two other *summary* abstract objects ($a_{>i}$ and $b_{>i}$). Array elements `a[i]` and `b[i]` are not grouped with any other array elements, and are represented by two *non-summary* abstract objects (a_i and b_i). Such partitioning allows the analysis to perform a strong update when it interprets the assignment statement in the body of the loop.

Fig. 4.2(a) shows how the elements of both arrays are partitioned during the first iteration of the loop. The non-summary objects a_i and b_i represent array elements `a[0]` and `b[0]`, respectively. The value of an element represented by a_i ranges from -5 to 5 due to our initial assumption; thus, after interpreting the assignment `b[i]` $\leftarrow$ `a[i]` in the body of the loop, the analysis establishes that the value of the element represented by b_i also ranges from -5 to 5.

At the end of the iteration, as the induction variable `i` gets incremented, the grouping of concrete array elements changes. The elements `a[0]` and `b[0]` move into the groups of concrete array elements that are represented by $a_{<i}$ and $b_{<i}$, respectively; thus, objects $a_{<i}$ and $b_{<i}$ inherit the numeric properties that have been synthesized for the objects a_i and b_i, respectively. The new abstract elements a_i and b_i, which now represent the array elements `a[1]` and `b[1]`, are constructed by *materializing* (with the use of the *expand* operation—see §3.3.4) two objects from the corresponding groups of concrete array elements represented by $a_{>i}$ and $b_{>i}$, respectively. Fig. 4.2(b) shows how the arrays `a` and `b` are partitioned during the second iteration. The net result, at the end of the first iteration, is that the analysis has established that the value of the concrete array element represented by $b_{<i}$ ranges from -5 to 5.

On the second iteration, the situation repeats with one exception: at the end of the iteration, when the array elements are again repartitioned, the new numeric properties for the abstract objects $a_{<i}$ and $b_{<i}$ are constructed by *merging* (with the use of the *fold* operation—see §3.3.3) the current

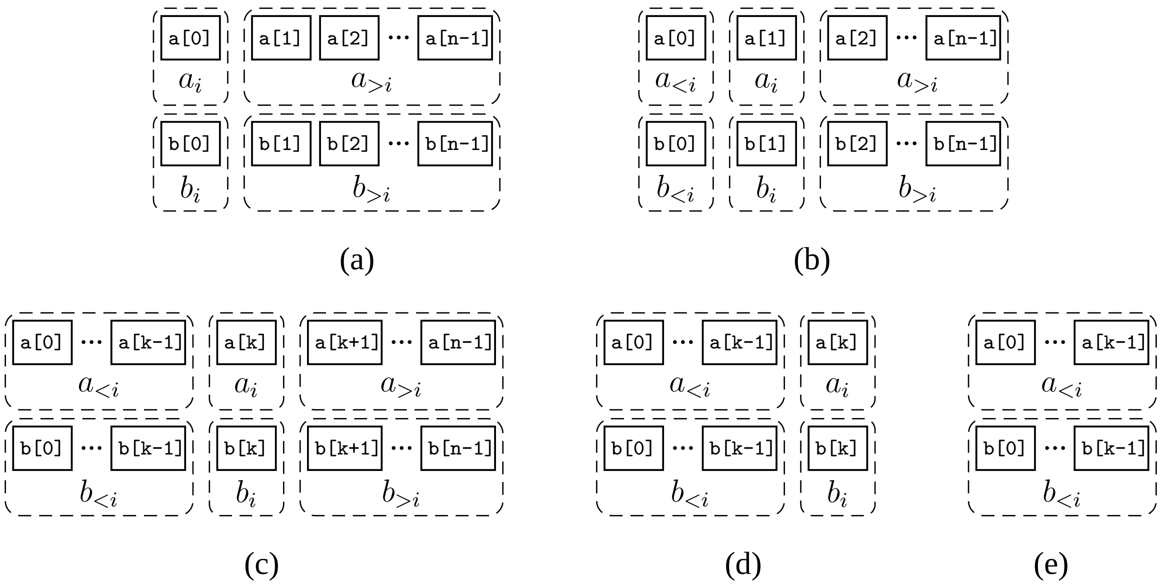


Figure 4.2 Partitionings of array elements at different points in the execution of the array-copy function: (a) on the 1-st loop iteration; (b) on the 2-nd iteration; (c) on the k-th iteration; (d) on the last iteration; (e) after exiting the loop.

numeric properties for $a_{<i}$ and $b_{<i}$ with the numeric properties that have been synthesized for the objects a_i and b_i, respectively. Note that because the values of the array elements that were represented by $b_{<i}$ range from -5 to 5 (the analysis established that at the end of the first iteration), and the value of the element that is represented by b_i also ranges form -5 to 5 (due to the assignment `b[i]` $\leftarrow$ `a[i]`), the analysis concludes that the values of the array elements represented by $b_{<i}$ after the second iteration also range from -5 to 5.

As the value of variable `i` increases with each iteration, more and more of the concrete array elements of both arrays move from the two groups represented by objects $a_{>i}$ and $b_{>i}$, to the two groups represented by objects a_i and b_i, and finally, to the two groups represented by objects $a_{<i}$ and $b_{<i}$. Fig. 4.2(c) shows how the arrays are partitioned on the $k+1$-st iteration. The concrete array elements that are represented by $b_{<i}$ are the elements that have been initialized.

One can view this process of state-space exploration as performing an inductive argument. On each iteration, the numeric properties of the abstract element $b_{<i}$ are merged (*folded*) with the numeric properties of the abstract element b_i, which has been initialized on that iteration. As the result, the invariant that the values of array elements represented by $b_{<i}$ range from -5 to 5 is

maintained throughout the analysis. In this regard, the analysis implements a form of *inductive reasoning*.

An important thing to observe is that, even though the array partitions shown in Fig. 4.2 (b) and (c) describe different groupings of concrete array elements, both partitions have the same sets of abstract array elements. Therefore, from the point of view of the analysis these partitions are the same. To establish which concrete array elements are represented by a particular abstract element, the analysis directly models the indices of array elements in the numeric state associated with each partition.

Fig. 4.2(e) shows how the array elements are partitioned after exiting from the loop: all elements of both arrays are represented by $a_{<i}$ and $b_{<i}$, respectively. As we have just shown above, the analysis is able to establish that the values of the concrete array elements represented by $b_{<i}$ at the end of each iteration ranged from -5 to 5. Thus, after exiting the loop, the analysis concludes that (i) all elements of array `b` were initialized, and (ii) the values stored in array `b` range from -5 to 5.

In practice, for the program in Fig. 4.1, a more general property may be of interest, such as "the value of each element of array `b` is equal to the value of the element of array `a` with the same index". However, this property is beyond the capabilities of the summarizing abstraction.[2] To capture such properties, we augment each abstract object in the representation of array `b` with an extra value that indicates whether the property holds for (i) all, (ii) some, or (iii) none of the concrete array elements represented by that abstract object. Formally, we do it by introducing an auxiliary three-valued unary predicate δ, which evaluates to the values 1, $1/2$, and 0 on the abstract elements of array `b` to represent cases (i), (ii), and (iii), respectively.

[2] It may be argued that this property is, in fact, *universal*. Let A and B denote sets of elements of arrays `a` and `b`, respectively. The property can be expressed as follows:

$$\forall a \in A\ \forall b \in B\ [\, a.\mathit{index} = b.\mathit{index} \ \Rightarrow\ a.\mathit{value} = b.\mathit{value}\,]\,,$$

and thus, a summarizing abstraction should be able to capture it. However, note that to capture this property, a summarizing abstraction must use a numeric abstract domain that is able to represent implication (see §3.5); however, it is extremely rare for a numeric domains to be capable of representing implication.

The analysis proceeds as follows: the abstract objects that represent the elements of array `b` start with δ having the value $1/2$, which indicates that the analysis has no knowledge about the values stored in array `b`. On each iteration, the property is established for the array element represented by b_i; i.e., $\delta(b_i)$ is set to 1. At the end of each iteration, the new value for $\delta(b_{<i})$ is obtained by joining the current value for $\delta(b_{<i})$ with $\delta(b_i)$: on the first iteration, the object $b_{<i}$ does not exist, so $\delta(b_{<i})$ is set to be equal to $\delta(b_i)$, which is equal to 1; on the following iterations, the new value for $\delta(b_{<i})$ is determined by joining its current value, which is 1, with the value $\delta(b_i)$, which also equals to 1. Thus, the analysis establishes that $\delta(b_{<i})$ is 1 after each iteration, which indicates that the property holds for all initialized array elements.

4.2 Concrete semantics

In this section, we extend the concrete program semantics from §2.1 to support a fixed, finite number of arrays. Understandably, the structure of the program state will become more complicated: in addition to the function that maps program variables to the corresponding values, we add a corresponding function for each array. These functions will map each array element of the corresponding array to a pair of numeric quantities: the value of the element and its index. Also, we add two new program-state transitions: one for reading an array element and one for writing an array element.

4.2.1 Concrete Program States

We denote the set of scalar variables and the set of arrays used in the program by

$$\mathcal{S} = \{v_1, ..., v_n\} \quad \text{and} \quad \mathcal{A} = \{A_1, ..., A_k\}\,,$$

respectively. These sets will be the same across all concrete program states that may arise as a result of program execution. Note that the set $\mathcal{S}$ directly corresponds to the set of variables *Vars*, which we defined in Chapter 2. The set of elements of each array may differ from state to state. We will use the notation A_S to denote the set of elements of array $A \in \mathcal{A}$ in program state S. To ensure proper sequencing of array elements, we assume that a concrete state explicitly assigns to each array element its proper index position in the corresponding array.

Each concrete program state S is encoded with the following set of functions:

- $\textit{Value}_S : \mathcal{S} \rightarrow \mathbb{V}$ maps each scalar variable to its corresponding value. These functions are similar to the functions that we used to define the program states in Chapter 2. In fact, all of the program constructs that are not related to arrays (i.e., numeric expressions, conditional expressions, assignment transitions, and assume transitions), operate on these functions in the exact same way as was defined in Chapter 2.

- $\textit{Element}_S^A : A_S \rightarrow \mathbb{V}^2$ is a family of functions (one for each array in $\mathcal{A}$) that maps the elements of a given array $A \in \mathcal{A}$ to a pair of quantities: the value of the element and its index.

Example 4.1 Suppose that a program operates on two scalar variables, `i` and `j`, and an array `B` of size 10. Suppose that at some point in the execution of the program, the values of variables `i` and `j` are 4 and 7, respectively, and the values that are stored in array `B` are $\{1, 3, 8, 12, 5, 7, 4, -2, 15, 6\}$. We encode the concrete state S of the program as follows:

$$\mathcal{S} = \{i, j\}, \quad \mathcal{A} = \{B\}, \quad B_S = \{b_0, \ldots, b_9\}$$

$$\textit{Value}_S = [i \mapsto 4, j \mapsto 7]$$

$$\textit{Element}_S^B = [b_0 \mapsto \langle 1, 0\rangle,\ b_1 \mapsto \langle 3, 1\rangle,\ b_2 \mapsto \langle 8, 2\rangle,\ \ldots,\ b_9 \mapsto \langle 6, 9\rangle]$$

To simplify the presentation, we define functions $\textit{Value}_S^A$ and $\textit{Index}_S^A$ with the signature $A_S \rightarrow \mathbb{V}$ to retrieve the value and the index of a given element of array A, respectively:

$$\textit{Value}_S^A(a) = \textit{Element}_S^A(a)[1] \qquad \text{and} \qquad \textit{Index}_S^A(a) = \textit{Element}_S^A(a)[2]$$

4.2.2 Array Transitions

We extend the set of program-state transitions with transitions for reading and writing array elements. The transition $v_1 \leftarrow A\,[v_2]$, where $v_1, v_2 \in \mathcal{S}$ and $A \in \mathcal{A}$, reads the value of the element of array A whose index is specified by the value of variable v_2, and assigns that value to variable v_1. The $A\,[v_2] \leftarrow v_1$ assigns the value of variable v_1 to the element of array A whose index is specified by the value of variable v_2. Let S denote an arbitrary program state. The concrete semantics for an array-read transition is specified as follows:

$$[\![v_1 \leftarrow A\,[v_2]]\!](S) = S' \quad \text{s.t.} \quad \left[\begin{array}{c} \exists u \in A_S \ \left[\mathit{Index}_S^A(u) = \mathit{Value}_S(v_2) \right] \\ \mathit{Value}_{S'}(v) = \begin{cases} \mathit{Value}_S^A(u) & \text{if } v = v_1 \\ \mathit{Value}_S(v) & \text{otherwise} \end{cases} \\ \forall B \in \mathcal{A} \ \ \forall b \in B \ \left[\mathit{Element}_{S'}^B(b) = \mathit{Element}_S^B(b) \right] \end{array} \right]$$

The concrete semantics for an array-write transition is given by:

$$[\![A\,[v_2] \leftarrow v_1]\!](S) = S' \quad \text{s.t.} \quad \left[\begin{array}{c} \exists u \in A_S \ \left[\mathit{Index}_S^A(u) = \mathit{Value}_S(v_2) \right] \\ \forall v \in \mathcal{S} \ \left[\mathit{Value}_{S'}(v) = \mathit{Value}_S(v) \right] \\ \forall B \in \mathcal{A} \setminus \{A\} \ \ \forall b \in B \ \left[\mathit{Element}_{S'}^B(b) = \mathit{Element}_S^B(b) \right] \\ \mathit{Element}_{S'}^A(a) = \begin{cases} \langle \mathit{Value}_S(v_1), \mathit{Value}_S(v_2) \rangle & \text{if } a = u \\ \mathit{Element}_S^A(a) & \text{otherwise} \end{cases} \end{array} \right]$$

The semantics for the transformers is defined in a straightforward way (the definitions primarily ensure that the part of the state that is not modified is preserved). Note, however, that the above transformers are *partial*: that is, if the condition

$$\exists u \in A_S \ \left[\mathit{Index}_S^A(u) = \mathit{Value}_S(v_2) \right]$$

is not satisfied, the result of either transformer is undefined. It can be easily seen that this condition implements an *out-of-bounds* check for the array access. That is, we assume that if an out-of-bounds array access occurs during program execution, the program terminates.

4.3 Array Abstraction

In this section, we show how the sets of concrete program states are abstracted. Each abstract program state $S^\sharp$ is a function $S^\sharp : \mathcal{P} \rightarrow \mathcal{N} \times \mathcal{X}$, where $\mathcal{P}$ denotes the space of possible array partitions, $\mathcal{N}$ denotes the space of possible numeric states, and $\mathcal{X}$ denotes space of possible valuations of auxiliary predicates. In the next few sections, we define what each of these spaces look like. We will refer to a triple $\left\langle P^\sharp, \Omega^\sharp, \Delta^\sharp \right\rangle$, where $P^\sharp \in \mathcal{P}$, $\Omega^\sharp \in \mathcal{N}$, $\Delta^\sharp \in \mathcal{X}$, and $S^\sharp(P^\sharp) = \left\langle \Omega^\sharp, \Delta^\sharp \right\rangle$, as an *abstract memory configuration*.

The standard operations for the abstract state are defined point-wise, that is, for all $P^\sharp \in \mathcal{P}$

$$(S_1^\sharp \sqcup S_2^\sharp)(P^\sharp) = S_1^\sharp(P^\sharp) \sqcup S_2^\sharp(P^\sharp), \quad \text{where} \quad \left\langle \Omega_1^\sharp, \Delta_1^\sharp \right\rangle \sqcup \left\langle \Omega_2^\sharp, \Delta_2^\sharp \right\rangle = \left\langle \Omega_1^\sharp \sqcup \Omega_2^\sharp, \Delta_1^\sharp \sqcup \Delta_2^\sharp \right\rangle$$

and

$$(S_1^\sharp \sqcap S_2^\sharp)(P^\sharp) = S_1^\sharp(P^\sharp) \sqcap S_2^\sharp(P^\sharp), \quad \text{where} \quad \left\langle \Omega_1^\sharp, \Delta_1^\sharp \right\rangle \sqcap \left\langle \Omega_2^\sharp, \Delta_2^\sharp \right\rangle = \left\langle \Omega_1^\sharp \sqcap \Omega_2^\sharp, \Delta_1^\sharp \sqcap \Delta_2^\sharp \right\rangle.$$

The partial order is defined as

$$S_1^\sharp \sqsubseteq S_2^\sharp \quad \triangleq \quad \forall P^\sharp \in \mathcal{P} \left[S_1^\sharp(P^\sharp) \sqsubseteq S_2^\sharp(P^\sharp) \right],$$

where

$$\left\langle \Omega_1^\sharp, \Delta_1^\sharp \right\rangle \sqsubseteq \left\langle \Omega_2^\sharp, \Delta_2^\sharp \right\rangle \quad \triangleq \quad \Omega_1^\sharp \sqsubseteq \Omega_2^\sharp \ \wedge \ \Delta_1^\sharp \sqsubseteq \Delta_2^\sharp.$$

Note that some array partitions may not arise (and, in fact, will not arise) in the analysis of the program. We assume that the abstract state maps the partitions that did not arise to the tuple $\langle \bot_{\mathcal{N}}, \bot_{\mathcal{X}} \rangle$.

In the next three sections, we will show how, given a *single* concrete state, to construct the abstract state that represents it. The abstraction for a set of concrete states is constructed by joining the abstractions of the individual concrete states in the set.

4.3.1 Array Partitioning

The goals of array partitioning are two-fold. First, we would like to isolate in separate groups the array elements that are assigned to. This allows the analysis to perform strong updates when assigning to these elements. Second, we would like to group elements with similar properties together to minimize the precision loss due to summarization.

In this thesis, we propose an array-partitioning scheme based on numeric relationships among indices of array elements and values of scalar variables. In particular, given a set of scalar variables, we partition an array so that each element whose index is equal to the value of any of the variables in the set is placed in a group by itself. Such groups are represented by *non-summary* abstract array elements. The consecutive array segments in-between the indexed elements are grouped together. Such groups are represented by *summary* abstract array elements. We will refer to this partitioning scheme as *linear array partitioning*.

We define array partitions by using a fixed set of *partitioning functions*, denoted by Π. Each function $\rho \in \Pi$ is parametrized by an array and a single scalar variable. Let $A \in \mathcal{A}$ and $v \in \mathcal{S}$. In a concrete state S, a function $\rho_{A,v}$ is interpreted as:

$$\rho_{A,v} : A_S \to \{-1, 0, 1\},$$

and is evaluated as follows:

$$\rho_{A,v}(u) = \begin{cases} -1 & \text{if } \mathit{Index}^A_S(u) < \mathit{Value}_S(v) \\ 0 & \text{if } \mathit{Index}^A_S(u) = \mathit{Value}_S(v) \\ 1 & \text{if } \mathit{Index}^A_S(u) > \mathit{Value}_S(v) \end{cases}$$

The choice of values is completely arbitrary as long as the function evaluates to a different value for each of the three cases. We denote the set of partitioning functions parameterized by array A by Π_A.

In a given concrete state, we partition each array $A \in \mathcal{A}$ by grouping together elements of A for which *all* partitioning functions in Π_A evaluate to the same values. Each group is represented by an abstract array element: a *non-summary* element, if at least one partitioning function evaluates to 0 for the array elements in the group; a *summary* element, otherwise. If the set Π_A is empty, all of the elements of array A are grouped together into a single summary element.

The values to which partitioning functions evaluate on the array elements in a group uniquely determine the abstract element that is used to represent that group. We will continue to use the intuitive abstract-element naming introduced in §4.1, e.g., the summary abstract array element $b_{>i,<j}$ represents the group of array elements whose indices are greater than the value of variable `i`, but less than the value of variable `j`.

Formally, array partition $P^\sharp$ maps each array in $\mathcal{A}$ to a corresponding set of abstract array elements. We say that two array partitions are equal if they map all arrays in $\mathcal{A}$ to the same sets:

$$P_1^\sharp = P_2^\sharp \quad \triangleq \quad \forall A \in \mathcal{A} \, \left[P_1^\sharp(A) = P_2^\sharp(A) \right]$$

The following example illustrates array partitioning.

Example 4.2 Assume the same situation as in Ex. 4.1. Let the set of partitioning functions Π be $\{\rho_{B,i}, \rho_{B,j}\}$. The elements of array B are partitioned into five groups, each of which is represented by an abstract array element:

	Concrete Elements	Abstract Element	Summary
(i)	$\{b_0, b_1, b_2, b_3\}$	$b_{<i,<j}$	$\star$
(ii)	$\{b_4\}$	$b_{i,<j}$	
(iii)	$\{b_5, b_6\}$	$b_{>i,<j}$	$\star$
(iv)	$\{b_7\}$	$b_{>i,j}$	
(v)	$\{b_8, b_9\}$	$b_{>i,>j}$	$\star$

The $\star$ in the last column marks summary abstract array elements. Thus,

$$P^\sharp = [B \mapsto \{b_{<i,<j},\ b_{i,<j},\ b_{>i,<j},\ b_{>i,j},\ b_{>i,>j}\}]$$

Note, that each abstract element of array A corresponds to a valuation of partitioning functions in Π_A: there are $3^{|\Pi_A|}$ possible valuations. However, not all valuations of partitioning functions are consistent with the respect to the array structure. In fact, due to linear nature of partitioning, there can be at most $2 \times |\Pi_A| + 1$ abstract array elements. Still, the number of possible array partitions (i.e., the number of sets of abstract array elements that represent the corresponding array in a consistent manner) is combinatorially large. However, our observations show that only a small fraction of these partitions actually occur in practice.

The approach that is presented in this section illustrates only one of the possibilities for partitioning an array. We found this partitioning to be useful when consecutive array elements share similar properties, e.g., when analyzing simple array-initialization loops and simple array-sorting algorithms, which constitute a large portion of actual uses of arrays. However, in more complicated examples, e.g., when using double indexing to initialize an array and using an array to store a complex data structure (such as a tree), the above array partitioning is not likely to succeed. This issue remains to be addressed in the future.

4.3.2 Numeric Abstraction

We represent the numeric state associated with an array partition with an element of a summarizing abstract domain, which we described in Chapter 3. In this section, we show how to construct the corresponding domain element for a given concrete state.

Conceptually, we apply the *partial* summarizing abstraction to each array individually (we use the extension that handles multiple values from §3.6). For an array $A \in \mathcal{A}$, the concrete universe U_A corresponds to A_S, and the abstract universe $U_A^\sharp$ corresponds to $P^\sharp(A)$. The concrete state is given by the function $\mathit{Element}_S^A : A_S \to \mathbb{V}^2$. The set of fields is $F = \{\mathit{value}, \mathit{index}\}$. The function $\pi_S : A_S \to P^\sharp(A)$ is defined naturally by the array-partitioning functions in Π_A.

The partial summarizing abstraction represents the function $\mathit{Element}_S^A : A_S \to \mathbb{V}^2$ by a set of functions with signature $P^\sharp(A) \to \mathbb{V}^2$ in the manner described in §3.2. This representation is further *flattened* to a set of functions with signatures $\hat{U^\sharp} \to \mathbb{V}$ in the manner described in §3.6, where $\hat{U^\sharp}$ denotes the *exploded* abstract universe:

$$\hat{U_A^\sharp} = \left\{u^\sharp.\mathit{value},\, u^\sharp.\mathit{index} \;\middle|\; u^\sharp \in P^\sharp(A)\right\}.$$

We denote the resulting set of functions by $\mathit{Element}_A^\sharp \in \wp(\hat{U_A^\sharp} \to \mathbb{V})$.

In the next abstraction step, we merge together the sets of functions $\mathit{Element}_A^\sharp$ for each array $A \in \mathcal{A}$, and the function $\mathit{Value}_S : \mathcal{S} \to \mathbb{V}$, which maps scalar variables to corresponding values, to form a set of functions Ω with the domain:

$$\hat{U^\sharp} = \mathcal{S} \cup \bigcup_{A \in \mathcal{A}} \hat{U_A^\sharp}.$$

This set of functions is then represented with a $\left|\hat{U^\sharp}\right|$-dimensional element of some existing numeric abstract domain. The choice of the domain depends on the particular properties the analysis needs to establish. In the rest of the chapter, we assume that the polyhedral abstract domain is used.

The numeric state is manipulated by the operations and abstract transformers that we defined in Chapter 3. To apply a particular transformer, the scalar variables and array elements are duly remapped to the corresponding dimensions in the abstract domain.

Example 4.3 Assume the same situation as in Ex. 4.2. Fig. 4.3 illustrates individual steps in the abstraction of the numeric state. The concrete state is shown at the top. The tables represent sets of functions (functions correspond to the rows in the table). We abbreviate the names of fields *value* and *index* by v and i, respectively. The set of functions $\mathit{Element}_A^\sharp$ is not shown, but it is similar to the set Ω with the first two columns (for i and j) taken out. The resulting set of functions Ω is abstracted by a 12-dimensional polyhedron, specified by the following set of linear constraints:

$$i = 4, \quad j = 7,$$

$$b_{<i,<j}.i + 1 \leq b_{<i,<j}.v, \quad 3 \times b_{<i,<j}.v \leq 11 \times b_{<i,<j}.i + 3, \quad 2 \times b_{<i,<j}.v \geq 9 \times b_{<i,<j}.i + 3,$$

$$b_{i,<j}.i = 4, \quad b_{i,<j}.v = 5,$$

$$5 \leq b_{>i,<j}.v \leq 6, \quad b_{>i,<j}.v = 22 - 3 \times b_{>i,<j}.i,$$

$$b_{>i,j}.i = 7, \quad b_{>i,j}.v = -2,$$

$$6 \leq b_{>i,>j}.v \leq 15, \quad b_{>i,>j}.v = 87 - 9 \times b_{>i,>j}.i$$

We show the constraints for each abstract array element on a separate line. Note that, for non-summary element $b_{i,<j}$ and $b_{>i,j}$, the values and the indices correspond to those of the concrete elements that they represent. Also note that, for summary elements, some relationships between the indices and values of the concrete elements are retained.

4.3.3 Beyond summarizing domains

Summarizing numeric domains can be used to reason about universal numeric properties of summarized array elements. However, the relationships among values of objects that are summarized together are lost. This precludes a summarizing numeric abstraction from being able to express certain properties of interest, e.g., it is impossible to express the fact that a set of array

Values_S :

i	j
4	7

$\mathit{Element}^A_S$:

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9
$\langle 1,0\rangle$	$\langle 3,1\rangle$	$\langle 8,2\rangle$	$\langle 12,3\rangle$	$\langle 5,4\rangle$	$\langle 7,5\rangle$	$\langle 4,6\rangle$	$\langle -2,7\rangle$	$\langle 15,8\rangle$	$\langle 6,9\rangle$

$(\alpha_2 \circ \alpha_1)(\mathit{Element}^A_S)$:

$b_{<i,<j}$	$b_{i,<j}$	$b_{>i,<j}$	$b_{>i,j}$	$b_{>i,>j}$
$\langle 1,0\rangle$	$\langle 5,4\rangle$	$\langle 7,5\rangle$	$\langle -2,7\rangle$	$\langle 15,8\rangle$
$\langle 1,0\rangle$	$\langle 5,4\rangle$	$\langle 7,5\rangle$	$\langle -2,7\rangle$	$\langle 6,9\rangle$
$\langle 1,0\rangle$	$\langle 5,4\rangle$	$\langle 4,6\rangle$	$\langle -2,7\rangle$	$\langle 15,8\rangle$
$\langle 1,0\rangle$	$\langle 5,4\rangle$	$\langle 4,6\rangle$	$\langle -2,7\rangle$	$\langle 6,9\rangle$
$\langle 3,1\rangle$	$\langle 5,4\rangle$	$\langle 7,5\rangle$	$\langle -2,7\rangle$	$\langle 15,8\rangle$
$\langle 3,1\rangle$	$\langle 5,4\rangle$	$\langle 7,5\rangle$	$\langle -2,7\rangle$	$\langle 6,9\rangle$
$\langle 3,1\rangle$	$\langle 5,4\rangle$	$\langle 4,6\rangle$	$\langle -2,7\rangle$	$\langle 15,8\rangle$
$\langle 3,1\rangle$	$\langle 5,4\rangle$	$\langle 4,6\rangle$	$\langle -2,7\rangle$	$\langle 6,9\rangle$
$\langle 8,2\rangle$	$\langle 5,4\rangle$	$\langle 7,5\rangle$	$\langle -2,7\rangle$	$\langle 15,8\rangle$
$\langle 8,2\rangle$	$\langle 5,4\rangle$	$\langle 7,5\rangle$	$\langle -2,7\rangle$	$\langle 6,9\rangle$
$\langle 8,2\rangle$	$\langle 5,4\rangle$	$\langle 4,6\rangle$	$\langle -2,7\rangle$	$\langle 15,8\rangle$
$\langle 8,2\rangle$	$\langle 5,4\rangle$	$\langle 4,6\rangle$	$\langle -2,7\rangle$	$\langle 6,9\rangle$
$\langle 12,3\rangle$	$\langle 5,4\rangle$	$\langle 7,5\rangle$	$\langle -2,7\rangle$	$\langle 15,8\rangle$
$\langle 12,3\rangle$	$\langle 5,4\rangle$	$\langle 7,5\rangle$	$\langle -2,7\rangle$	$\langle 6,9\rangle$
$\langle 12,3\rangle$	$\langle 5,4\rangle$	$\langle 4,6\rangle$	$\langle -2,7\rangle$	$\langle 15,8\rangle$
$\langle 12,3\rangle$	$\langle 5,4\rangle$	$\langle 4,6\rangle$	$\langle -2,7\rangle$	$\langle 6,9\rangle$

Ω :

		$b_{<i,<j}$		$b_{i,<j}$		$b_{>i,<j}$		$b_{>i,j}$		$b_{>i,>j}$	
i	j	v	i	v	i	v	i	v	i	v	i
4	7	1	0	5	4	7	5	-2	7	15	8
4	7	1	0	5	4	7	5	-2	7	6	9
4	7	1	0	5	4	4	6	-2	7	15	8
4	7	1	0	5	4	4	6	-2	7	6	9
4	7	3	1	5	4	7	5	-2	7	15	8
4	7	3	1	5	4	7	5	-2	7	6	9
4	7	3	1	5	4	4	6	-2	7	15	8
4	7	3	1	5	4	4	6	-2	7	6	9
4	7	8	2	5	4	7	5	-2	7	15	8
4	7	8	2	5	4	7	5	-2	7	6	9
4	7	8	2	5	4	4	6	-2	7	15	8
4	7	8	2	5	4	4	6	-2	7	6	9
4	7	12	3	5	4	7	5	-2	7	15	8
4	7	12	3	5	4	7	5	-2	7	6	9
4	7	12	3	5	4	4	6	-2	7	15	8
4	7	12	3	5	4	4	6	-2	7	6	9

Figure 4.3 Numeric abstraction for the program state in Ex. 4.1.

elements that are summarized together are in sorted order. In Ex. 4.3, the resulting numeric state is only able to capture the property that the values of the concrete array elements represented by $b_{(<i,<j)}$ range from 1 to 12, but not that those elements are sorted in ascending order.

To capture properties that are beyond the capabilities of summarizing numeric domains, we introduce a set of auxiliary predicates, denoted by Δ. In a concrete state S, a predicate in $\delta_A \in \Delta$ maps each element of array A to a boolean value: to 1 if the property of interest holds for that element, and to 0 otherwise:

$$\delta_A : A^S \rightarrow \{0,1\}\,.$$

The semantics of an auxiliary predicate is specified via a formula that is evaluated in the concrete state.

When the elements of array A are summarized, we *join* the corresponding values of δ_A in a *3-valued logic lattice* [100]. In 3-valued logic, an extra value, denoted by $1/2$, is added to the set of Boolean values $\{0,1\}$. The order is defined as follows:

$$l_1 \sqsubseteq l_2 \text{ iff } l_1 = l_2 \text{ or } l_2 = 1/2$$

Thus,

$$1/2 \sqcup 0 = 1/2 \sqcup 1 = 0 \sqcup 1 = 1/2.$$

The resulting value is attached to the corresponding abstract array element.

In an abstract memory configuration, we use an abstract counterpart of the predicate, denoted by $\delta_A^\sharp$, to map abstract array elements to corresponding values:

$$\delta_A^\sharp : P^\sharp(A) \rightarrow \{0,1,1/2\}$$

Let $u \in P^\sharp(A)$ be an arbitrary abstract array element. The value of $\delta_A^\sharp(u)$ is interpreted as follows: the value 1 indicates that the property holds for all of the array elements represented by u; the value 0 indicates that the property does not hold for any of the array elements represented by u; and the value $1/2$ indicates that property may hold for some of the array elements represented by u, but may not hold for the rest of the elements.

Example 4.4 Assume the same situation as in Ex. 4.3. We introduce a predicate δ_B that evaluates to 1 for array elements that are in ascending order, and to 0 for the elements that are not:

$$\delta_B(u) \quad \triangleq \quad \forall t \in B \ \left[\textit{Index}_S^B(t) < \textit{Index}_S^B(u) \ \Rightarrow \ \textit{Value}_S^B(t) \leq \textit{Value}_S^B(u) \right],$$

where variable u is a free variable that binds to an array element when the predicate is evaluated. In the concrete state shown in Ex. 4.1, δ_B evaluates to 1 for the elements b_0, b_1, b_2, b_3, and b_8; and to 0 for the remaining elements. The values associated with the abstract array elements are constructed as follows:

$$\begin{array}{llll}
\delta_B^\sharp(b_{<i,<j}) & = \delta_B(b_0) \sqcup \delta_B(b_1) \sqcup \delta_B(b_2) \sqcup \delta_B(b_3) & = 1 \sqcup 1 \sqcup 1 \sqcup 1 & = 1 \\
\delta_B^\sharp(b_{i,<j}) & = \delta_B(b_4) & & = 0 \\
\delta_B^\sharp(b_{>i,<j}) & = \delta_B(b_5) \sqcup \delta_B(b_6) & = 0 \sqcup 0 & = 0 \\
\delta_B^\sharp(b_{>i,j}) & = \delta_B(b_7) & & = 0 \\
\delta_B^\sharp(b_{>i,>j}) & = \delta_B(b_8) \sqcup \delta_B(b_9) & = 1 \sqcup 0 & = 1/2
\end{array}$$

The part of an abstract memory configuration that stores the interpretation of auxiliary predicates is denoted by $\Delta^\sharp \in \mathcal{X}$ and is defined as:

$$\Delta^\sharp(\delta_A, u) = \delta_A^\sharp(u)$$

We define a partial-order relation for interpretations of auxiliary predicates as follows:

$$\Delta_1^\sharp \sqsubseteq \Delta_2^\sharp \quad \triangleq \quad \forall A \in \mathcal{A} \ \ \forall \delta_A \in \Delta \ \ \forall u \in P^\sharp(A) \ \left[\Delta_1^\sharp(\delta_A, u) \sqsubseteq \Delta_2^\sharp(\delta_A, u) \right].$$

The join operation for interpretations of auxiliary predicates is defined as follows: we say that $\Delta_1^\sharp \sqcup \Delta_2^\sharp = \Delta^\sharp$, where for all $A \in \mathcal{A}$, for all $\delta_A \in \Delta$, and for all $u \in P^\sharp(A)$

$$\Delta^\sharp(\delta_A, u) = \Delta_1^\sharp(\delta_A, u) \sqcup \Delta_2^\sharp(\delta_A, u).$$

4.4 Array Copy Revisited

In this section, we flesh out the schematic illustration of the analysis that was given in §4.1. The analysis is applied to the code shown in Fig. 4.1. We depict the abstract memory configurations that

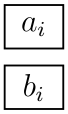
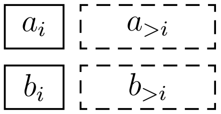
$P^\sharp$	$P^\sharp_1$: 	$P^\sharp_2$: 
$\Omega^\sharp$	$n = 1,\ i = 0,$ $a_i.index = 0$ $-5 \le a_i.value \le 5$ $b_i.index = 0$	$n \ge 2,\ i = 0,$ $a_i.index = b_i.index = 0$ $1 \le a_{>i}.index \le n-1$ $-5 \le a_i.value \le 5$ $-5 \le a_{>i}.value \le 5$ $1 \le b_{>i}.index \le n-1$
$\Delta^\sharp$	$\delta^\sharp_b = [b_i \mapsto 1/2]$	$\delta^\sharp_b = [b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$
	$\boldsymbol{S^\sharp_1}$	$\boldsymbol{S^\sharp_2}$

Figure 4.4 Abstract memory configurations (AMCs) that reach the head of the loop in Fig. 4.1 on the first iteration. The first AMC represents arrays of length 1; the second AMC represents arrays of length 2 and greater.

arise in the course of the analysis as follows. The partition of the arrays is shown graphically: solid boxes represent non-summary abstract array elements; dashed boxes represent summary abstract array elements. Numeric states are shown as sets of linear constraints. Auxiliary predicates are shown as maps from sets of abstract array elements to corresponding values in $\{0, 1, 1/2\}$.

Consider the program in Fig. 4.1. The set of scalar variables and the set of arrays are defined as follows: $\mathcal{S} = \{i, n\}$ and $\mathcal{A} = \{a, b\}$. The analysis uses the set of partitioning functions $\Pi = \{\rho_{a,i}, \rho_{b,i}\}$. As we suggested in §4.1, it is impossible for the summarizing abstraction to express the property "for every index k, the value of `b[k]` is equal to the value of `a[k]`". To capture this property, we introduce an auxiliary predicate δ_b, whose semantics is defined by

$$\delta_b(u) \quad \triangleq \quad \forall t \in a^S \ \left[\mathit{Index}^b_S(u) = \mathit{Index}^a_S(t) \ \Rightarrow \ \mathit{Value}^b_S(u) = \mathit{Value}^a_S(t) \right].$$

Fig. 4.4 shows the abstract state that reaches the head of the loop before the first iteration. The abstract state contains two abstract memory configurations: $S^\sharp_1$ and $S^\sharp_2$. Configuration $S^\sharp_1$ represents the case in which each array contains only a single element. Thus, each array is represented by a single abstract array element, a_i and b_i, respectively. The indices of both a_i and b_i are equal to zero, and the value of a_i ranges from -5 to 5.

Abstract memory configuration $S_2^\sharp$ represents the concrete states in which both arrays are of length greater than or equal to two. In this situation, each array is represented by two abstract elements: a_i and b_i represent the first elements of the corresponding arrays, while $a_{>i}$ and $b_{>i}$ represent the remaining elements. The numeric state associated with this partition indicates that the indices of the concrete array elements represented by a_i and b_i are equal to zero, the indices of the concrete array elements represented by $a_{>i}$ and $b_{>i}$ range from 1 to $n-1$, and the values of all concrete elements of array `a` range from -5 to 5.

The auxiliary predicate $\delta_b^\sharp$ evaluates to $1/2$ for all array elements in the abstract memory configurations $S_1^\sharp$ and $S_2^\sharp$. This means that, in the concrete states represented by $S_1^\sharp$ and $S_2^\sharp$, the values of the concrete elements of array `b` may either be equal to the values of the corresponding elements of array `a` or not.

Fig. 4.5 shows the abstract memory configurations that are accumulated at the head of the loop (in addition to the AMCs $S_1^\sharp$ and $S_2^\sharp$ shown in Fig. 4.4) on each iteration of the analysis. Fig. 4.6 shows the evolution, on successive iterations of the analysis, of the single abstract memory configuration that reaches the exit of the loop.

The analysis proceeds as follows. Both $S_1^\sharp$ and $S_2^\sharp$ satisfy the loop condition and are propagated into the body of the loop. After the assignment "`b[i]` $\leftarrow$ `a[i]`", two changes happen to both abstract memory configurations: (i) the constraint $a_i.value = b_i.value$ is added to their numeric states, and (ii) the value of auxiliary predicate $\delta_b^\sharp(b_i)$ is changed to 1.

At the end of the first iteration, as variable `i` is incremented, abstract memory configuration $S_1^\sharp$ is transformed into configuration $S_9^\sharp$ (shown in Fig. 4.6). The loop condition does not hold in $S_9^\sharp$; thus, this memory configuration is propagated to the program point ($\star\star$) at the exit of the loop. Abstract memory configuration $S_2^\sharp$ is transformed into two new abstract memory configurations $S_3^\sharp$ and $S_4^\sharp$. These memory configurations, along with $S_1^\sharp$ and $S_2^\sharp$, form the abstract state at the head of the loop at the beginning of the second iteration.

On the second iteration, the abstract memory configurations $S_3^\sharp$ and $S_4^\sharp$ are propagated through the assignment "`b[i]` $\leftarrow$ `a[i]`". As the result, their numeric states are updated to make $a_i.value = b_i.value$, and their valuations of auxiliary predicate $\delta_b^\sharp(b_i)$ are changed to 1.

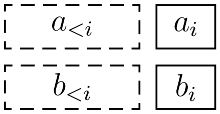
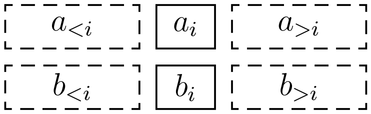
	$P^\sharp$	$P_3^\sharp$: 	$P_4^\sharp$:  $a_{<i}$ a_i $a_{>i}$ $b_{<i}$ b_i $b_{>i}$
2-nd iteration	$\Omega^\sharp$	$i = 1, n = 2,$ $a_i.index = b_i.index = 1$ $a_{<i}.index = 0$ $-5 \le a_i.value \le 5$ $-5 \le a_{<i}.value \le 5$ $b_{<i}.index = 0$ $b_{<i}.value = a_{<i}.value$	$i = 1, n \ge 3,$ $a_i.index = b_i.index = 1$ $2 \le a_{>i}.index \le n - 1$ $a_{<i}.index = 0$ $-5 \le a_i.value \le 5$ $-5 \le a_{>i}.value \le 5$ $-5 \le a_{<i}.value \le 5$ $2 \le b_{>i}.index \le n - 1$ $b_{<i}.value = a_{<i}.value$
	$\Delta^\sharp$	$\delta_b^\sharp = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$	$\delta_b^\sharp = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$
		$\boldsymbol{S_3^\sharp}$	$\boldsymbol{S_4^\sharp}$
3-rd iteration	$\Omega^\sharp$	$1 \le i \le 2$ $n = i + 1$ $a_i.index = i$ $0 \le a_{<i}.index \le i - 1$ $-5 \le a_i.value \le 5$ $-5 \le a_{<i}.value \le 5$ $b_i.index = i$ $0 \le b_{<i}.index \le i - 1$ $-5 \le b_{<i}.value \le 5$	$1 \le i \le 2$ $i = a_i.index = b_i.index$ $0 \le a_{<i}.index \le i - 1$ $i + 1 \le a_{>i}.index \le n - 1$ $-5 \le a_i.value \le 5$ $-5 \le a_{>i}.value \le 5$ $-5 \le a_{<i}.value \le 5$ $0 \le b_{<i}.index \le i - 1$ $i + 1 \le b_{>i}.index \le n - 1$ $-5 \le b_{<i}.value \le 5$
	$\Delta^\sharp$	$\delta_b^\sharp = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$	$\delta_b^\sharp = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$
		$\boldsymbol{S_5^\sharp}$	$\boldsymbol{S_6^\sharp}$
3-rd iteration (after widening)	$\Omega^\sharp$	$1 \le i$ $n = i + 1$ $a_i.index = i$ $0 \le a_{<i}.index \le i - 1$ $-5 \le a_i.value \le 5$ $-5 \le a_{<i}.value \le 5$ $b_i.index = i$ $0 \le b_{<i}.index \le i - 1$ $-5 \le b_{<i}.value \le 5$	$1 \le i$ $i = a_i.index = b_i.index$ $0 \le a_{<i}.index \le i - 1$ $i + 1 \le a_{>i}.index \le n - 1$ $-5 \le a_i.value \le 5$ $-5 \le a_{>i}.value \le 5$ $-5 \le a_{<i}.value \le 5$ $0 \le b_{<i}.index \le i - 1$ $i + 1 \le b_{>i}.index \le n - 1$ $-5 \le b_{<i}.value \le 5$
	$\Delta^\sharp$	$\delta_b^\sharp = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$	$\delta_b^\sharp = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$
		$\boldsymbol{S_7^\sharp}$	$\boldsymbol{S_8^\sharp}$

Figure 4.5 New abstract memory configurations (ACMs) that reach the head of the loop in Fig. 4.1 on the 2-nd and 3-rd iterations of the analysis. The last row shows the effect of widening on the numeric portion of abstract state: the numeric states obtained on the 2-nd iteration are widened with respect to the corresponding numeric states obtained on the 3-rd iteration.

$P^\sharp$		after 1-st iteration	after 2-nd iteration	after 3-rd iteration
$P_5^\sharp$: $a_{<i}$ $b_{<i}$	$\Omega^\sharp$	$n = 1, i = n,$ $a_{<i}.index = 0$ $-5 \leq a_{<i}.value \leq 5$ $b_{<i}.index = 0$ $b_{<i}.value = a_{<i}.value$	$1 \leq n \leq 2, i = n,$ $0 \leq a_{<i}.index \leq n-1$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$	$1 \leq n, i = n$ $0 \leq a_{<i}.index \leq n-1$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$
	$\Delta^\sharp$	$\delta_b^\sharp = [b_{<i} \mapsto 1]$	$\delta_b^\sharp = [b_{<i} \mapsto 1]$	$\delta_b^\sharp = [b_{<i} \mapsto 1]$
		$\boldsymbol{S_9^\sharp}$	$\boldsymbol{S_{10}^\sharp}$	$\boldsymbol{S_{11}^\sharp}$

Figure 4.6 The abstract state (consisting of a single abstract memory configuration) that reaches program point $(\star\star)$ after the 1-st, 2-nd, and 3-d iterations of the loop in Fig. 4.1. The last column shows the stabilized abstract state at $(\star\star)$.

At the end of the second iteration, the abstract memory configuration $S_3^\sharp$ is transformed into a configuration that has array partition $P_5^\sharp$. This abstract memory configuration is propagated to the program point $(\star\star)$ and is joined with $S_9^\sharp$ to yield $S_{10}^\sharp$. The abstract memory configuration $S_4^\sharp$ is transformed into two new abstract memory configurations, with array partitions $P_3^\sharp$ and $P_4^\sharp$. These configurations are propagated back to the head of the loop, and are joined with $S_3^\sharp$ and $S_4^\sharp$, resulting in abstract memory configurations $S_5^\sharp$ and $S_6^\sharp$, respectively. At this moment, the analysis extrapolates the loop behavior by widening the numeric state of $S_3^\sharp$ with respect to the numeric state of $S_5^\sharp$, and by widening the numeric state of $S_4^\sharp$ with respect to the numeric state of $S_6^\sharp$. The application of widening produces abstract memory configurations $S_7^\sharp$ and $S_8^\sharp$. Thus, the abstract state at the head of the loop at the beginning of the third iteration contains abstract memory configurations $S_1^\sharp$, $S_2^\sharp$, $S_7^\sharp$ and $S_8^\sharp$.

At the end of the third iteration, abstract memory configuration $S_7^\sharp$ is transformed into a configuration with array partition $P_5^\sharp$, which is propagated to the program point ($\star\star$), where it is joined with $S_{10}^\sharp$, resulting in $S_{11}^\sharp$. Abstract memory configuration $S_8^\sharp$ is transformed into two memory configurations, which are propagated to the head of the loop. However, those configurations are equivalent to $S_7^\sharp$ and $S_8^\sharp$, which were previously encountered by the analysis. Thus, at this stage, the abstract state at the head of the loop stabilizes and the analysis terminates.

The abstract state accumulated at the program point ($\star\star$) contains a single non-trivial memory configuration $S_{11}^{\sharp}$ (shown in Fig. 4.6). It is easy to see that this configuration represents only the concrete states in which (i) the values stored in the array `b` range from -5 to 5 (this follows from the constraint "$-5 \leq b_{<i} \leq 5$"in the numeric state); and (ii) the value of each element of array `b` is equal to the value of the element of array `a` with the same index (this follows from the fact that $\delta_b^{\sharp}(b_{<i})$ evaluates to the definite value 1).

4.5 Implementation of an Array-Analysis Tool

We built a prototype of our array analysis using the numeric extension that we implemented of TVLA tool [78] (which was described in §3.7). The implementation is based on the ideas that have been described above. However, to make the abstraction described in previous sections usable, we have to define the abstract counterparts for the concrete state transformers shown in §4.2.

In [31], it is shown that for a Galois connection defined by abstraction function α and concretization function γ, the best abstract transformer for a concrete transformer τ, denoted by $\tau^{\sharp}$, can be expressed as: $\tau^{\sharp} = \alpha \circ \tau \circ \gamma$. This defines the limit of precision obtainable using a given abstract domain; however, it is a non-constructive definition: it does not provide an *algorithm* for finding or applying $\tau^{\sharp}$.

For our tool, we defined *overapproximations* for the best abstract state transformers by using TVLA mechanisms. In the rest of this section, we give a brief overview of TVLA, and sketch the techniques for modeling arrays and defining abstract transformers.

4.5.1 Overview of TVLA

TVLA models concrete states by first-order logical structures. The elements of a structure's universe represent the concrete objects. Predicates encode relationships among the concrete objects. The abstract states are represented by *three-valued logical structures*, which are constructed by applying canonical abstraction to the sets of concrete states. The abstraction is performed by identifying a vector of unary predicates and representing the concrete objects for which these abstraction predicates evaluate to the same vector of values by a single element in the universe of a three-valued structure. In the rest of the paper, we refer to these abstract elements as *nodes*. A node that represents a single concrete object is called non-summary node, and a node that represent multiple concrete objects is called summary node.

TVLA distinguishes between two types of predicates: *core* predicates and *instrumentation* predicates. Core predicates are the predicates that are necessary to model the concrete states. Instrumentation predicates, which are defined in terms of core predicates, are introduced to capture properties that would otherwise be lost due to abstraction.

An abstract state transformer is defined in TVLA as a sequence of (optional) steps:

- A *focus step* replaces a three-valued structure by a set of more precise three-valued structures that represent the same set of concrete states as the original structure. Usually, focus is used to "materialize" a non-summary node from a summary node. The structures resulting from a focus step are not necessarily images of canonical abstraction, in the sense that they may have multiple nodes for which the abstraction predicates evaluate to the same values.

- A *precondition step* filters out the structures for which a specified property does not hold from the set of structures produced by focus. Generally, preconditions are used to model conditional statements.

- An *update step* transforms each structure that satisfies the precondition, to reflect the effect of an assignment statement. This is done by creating a new structure in which the core and instrumentation predicates are assigned appropriate (i.e., sound) values.

- A *coerce step* is a cleanup operation that "sharpens" updated three-valued structures by making them comply with a set of globally defined integrity constraints.

- A *blur step* restores the "canonicity" of coerced three-valued structures by applying canonical abstraction to them, i.e., merging together the nodes for which the abstraction predicates evaluate to the same values.

In §3.7, we extended TVLA with the capability to explicitly model numeric quantities. In particular, we added the facilities to associate a set of numeric quantities with each concrete object, and equipped each three-valued logical structure with an element of a summarizing numeric domain to represent the values of these quantities in abstract states: i.e., each node in a three-valued structure is associated with a set of dimensions of a summarizing numeric domain. TVLA's specification language was extended to permit using numeric comparisons in logical formulas, and to specify numeric updates.

4.5.2 Modeling arrays

We encode concrete states of a program as follows. Each scalar variable and each array element corresponds to an element in the universe of the first-order logical structure. We also introduce a *core* unary predicate for each scalar variable and for each array. These predicates evaluate to 1 on the elements of the first-order structure that represent the corresponding scalar variable or the element of the corresponding array, and to 0 for the rest of the elements. Each element in the universe is associated with a numeric quantity that represents its value. Each array element is associated with an extra numeric quantity that represents its index position in the array.

To model the array structure in TVLA correctly, extra predicates are required. We model the adjacency relation among array elements by introducing a binary instrumentation predicate for each array. This predicate evaluates to 1 when evaluated on two adjacent elements of an array. To model the property that indices of array elements are contiguous and do not repeat, we introduce a unary instrumentation predicate for each array that encodes transitive closure of the adjacency relation.

Partitioning functions are defined by unary *instrumentation* predicates. Because a partitioning function may evaluate to three different values, whereas a predicate can only evaluate to 0 or 1, we use two predicates to encode each partitioning function. Auxiliary predicates from §4.3.3 are implemented as unary *instrumentation* predicates.

To perform the abstraction, we select a set of abstraction predicates that contains the predicates corresponding to scalar variables and arrays, the predicates that encode the transitive closure of adjacency relations for each array, and the predicates that implement the partitioning functions. The auxiliary predicates are non-abstraction predicates. The resulting three-valued structures directly correspond to the abstract memory configurations we defined in §4.3.

The transformers for the statements that do not require array repartitioning, e.g., conditional statements, assignments to scalar variables that are not used to index array elements, and array reads and writes are modeled in a straightforward way. The transformers for the statements that cause a change in array partitioning, i.e., updates of scalar variables that are used to index array elements, are defined as follows:

- first, *focus* is applied to the structure to materialize the array element that will be indexed by the variable after the update;
- then, the value of the scalar variable, and the interpretation of the partitioning predicates are updated;
- finally, *blur* is used to merge the array element that was indexed by the variable previously, into the appropriate summary node.

To update the interpretation of auxiliary predicates, the programmer must supply predicate-maintenance formulas for each statement that may change the values of those predicates. Also, to update the numeric state with the numeric properties encoded by the auxiliary as the grouping of the concrete array elements changes, a set of integrity constraints implied by the auxiliary predicates must be supplied.

Aside from the integrity constraints and update formulas for the auxiliary predicates, the conversion of an arbitrary program into a TVLA specification can be performed fully automatically. The remaining manual steps could also be automated by extending the techniques for *differencing* logical formulas, described in [95], with the capability to handle atomic numeric conditions. However, we did not pursue this direction in our research.

4.6 Experimental Evaluation

In this section, we describe the application of the analysis prototype to four simple examples, which encompass array manipulations that often occur in practice. We used a simple heuristic to obtain the set of partitioning functions for each array in the analyzed examples. In particular, for each array access “`a[i]`” in the program, we added a partitioning function $\pi_{a,i}$ to the set Π. This approach worked well for all of the examples, except for the insertion-sort implementation, which required the addition of an extra partitioning function.

4.6.1 Array initialization

Fig. 4.7(a) shows a piece of code that initializes array `a` of size `n`. Each array element is assigned a value equal to twice its index position in the array, plus 3. The purpose of this example is to illustrate that the analysis is able to automatically discover numeric constraints on the values of array elements.

The array-partitioning heuristic produces a single partitioning function $\pi_{a,i}$ for this example. The analysis establishes that after the code is executed, the values stored in the array range from 3 to $2 \times n + 1$. No human intervention in the form of introducing auxiliary predicates is required.

In contrast, other approaches that are capable of handling this example [43, 111] require that the predicate that specifies the expected bounds for the values of array elements be supplied explicitly, either by the user or by an automatic abstraction-refinement technique [73].

4.6.2 Partial array initialization

Fig. 4.7(b) contains a more complex array-initialization example. The code repeatedly compares elements of arrays `a` and `b` and, in case they are equal, writes their index position into the array `c`. The portion of array `c` that is initialized depends on the values stored in the arrays `a` and `b`. Three scenarios are possible: (i) none of the elements of `c` are initialized; (ii) an initial segment of `c` is initialized; (iii) all of `c` is initialized. The purpose of this example is to illustrate the handling of multiple arrays, as well as partial array initialization.

The array-partitioning heuristic derives a set of three partitioning functions for this example, one for each array: $\Pi = \{\pi_{a,i}, \pi_{b,i}, \pi_{c,j}\}$. The analysis establishes that, after the loop, the elements of array `c` with indices between 0 and $j - 1$ were initialized to values ranging from 0 to $n - 1$. Again, no auxiliary predicates are necessary.

```
int a[n], i, n;

i ← 0;
while(i < n) {
   a[i] ← 2 × i + 3;
   i ← i + 1;
}
```

(a)

```
int a[n], b[n], c[n], i, j, n;

i ← 0;
j ← 0;
while(i < n) {
   if(a[i] == b[i]) {
      c[j] ← i;
      j ← j + 1;
   }
   i ← i + 1;
}
```

(b)

```
void sort(int a[], int n) {
   int i, j, k, t;

   i ← 1;
   while(i < n) {
      j ← i;
      while(j > 0) {
         k ← j - 1;
         if(a[j] ≥ a[k]) break;

         t ← a[j];
         a[j] ← a[k];
         a[k] ← t;
         j ← j - 1;
      }
      i ← i + 1;
   }
}
```

(c)

Figure 4.7 Array manipulation code: (a) array-initialization loop; (b) partial array initialization; (c) insertion-sort routine.

The abstract state that reaches the exit of the loop contains four abstract memory configurations. The first configuration represents concrete states in which none of the array elements are initialized. The value of j, in this domain element, is equal to zero, and, thus, the array partition does not contain the abstract element $c_{<j}$.

The second and the third memory configurations represent the concrete states in which only an initial segment of array c is initialized. Two different memory configurations are required to represent this case because the analysis distinguishes the case of variable j indexing an element in the middle of the array from the case of j indexing the last element of the array.

The last abstract memory configuration represents the concrete states in which all elements of array c are initialized. In the concrete states represented by this memory configuration, the value of variable j is equal to the value of variable n, and all elements of array c are represented by the abstract array element $c_{<j}$.

The initialized array elements are represented by the abstract array element $c_{<j}$. The array partition of the first memory configuration does not contain element $c_{<j}$, which indicates that no

elements were initialized. The numeric states associated with the other abstract memory configurations capture the property that the values of initialized array elements range between 0 and $n-1$.

4.6.3 Insertion sort

Fig. 4.7(c) shows a procedure that sorts an array using insertion sort. Parameter `n` specifies the size of array `a`. The invariant for the outer loop is that the array is sorted up to the i-th element. The inner loop inserts the i-th element into the sorted portion of the array. An interesting detail about this implementation is that elements are inserted into the sorted portion of the array in reverse order. The purpose of this example is to demonstrate the application of the analysis to a more challenging problem.

The application of the array-partitioning heuristic yields $\Pi = \{\pi_{a,j}\}$. Unfortunately, this partitioning is not sufficient. We also need to use variable `i` to partition the array so that the sorted segment of the array is separate from the unsorted segment. However, because `i` is never explicitly used to index array elements, our array-partitioning heuristic fails to add $\pi_{a,i}$ to the set of partitioning functions. To successfully analyze this example, we have to manually add $\pi_{a,i}$ to Π.

Summarizing numeric domains are not able to preserve the order of summarized array elements. An auxiliary predicate, defined similarly to the predicate δ_B in Ex. 4.4, needs to be introduced. Our prototype implementation requires user involvement to specify the explicit update formulas for this predicate for each of the program statements. Fortunately, the majority of the program statements do not affect this predicate. Thus, the corresponding update formula for such statements is the identity function. The only non-trivial case is the assignment to an array element.

The human involvement necessitated by the analysis is (i) minor, and (ii) problem-specific. In particular, only one auxiliary predicate needs to be introduced. Furthermore, this predicate is not specific to a given implementation of a sorting algorithm. Rather, it can be reused in the analysis of other implementations, and even in the analysis of other sorting algorithms.

Example	Abstract Memory Configurations (AMCs)		Time for	Time
	Max AMCs per prog. pt.	Max nodes per AMC	Coerce & Focus (%)	(sec)
Array initialization	7	8	68.3	1.7
Partial initialization	35	20	86.3	194.0
Array copy	7	13	94.2	338.1
Insertion sort	38	14	85.5	48.5

Figure 4.8 Array-analysis measurements.

Also, this example identifies some directions for future research: (i) designing better techniques for the automatic array partitioning, and (ii) automatically discovering and maintaining auxiliary predicates.

4.6.4 Analysis Measurements

We ran the analysis prototype on an Intel-based Linux machine equipped with a 2.4 GHz Pentium 4 processor and 512Mb of memory. Fig. 4.8 shows the measurements we collected while analyzing the examples discussed above. We report the maximal number of abstract memory configurations encountered at a program point and the maximal number of abstract objects in an abstract memory configuration (these objects include scalar variables and abstract array elements). Also, we report the percentage of analysis time spent on performing TVLA's *focus* and *coerce* operations.

Overall, the analysis results show that our prototype implementation is able to establish interesting properties of array-manipulation code, but, at the same time, is not very efficient in terms of the analysis time: it takes our prototype on the order of minutes to analyze the simple examples we have considered. In principle, we believe that a dedicated implementation of the array analysis (i.e., not within the confines of some general framework, such as TVLA), as described in this chapter, will be much more effective in practice. Below we list some of the factors that make us believe so.

The analysis times are severely affected by our decision to implement the analysis prototype in TVLA. Because TVLA is a general framework, the structure of an array has to be modeled explicitly by introducing a number of instrumentation predicates and integrity constraints. Consequently, the majority of the analysis time is spent executing focus and coerce operations to ensure that the array structure is preserved. The measurements in Fig. 4.8 indicate that, on average, focus and coerce account for about 80% of the overall analysis time. Building a dedicated analysis implementation, in which the knowledge of the linear structure of arrays is built into the abstract state transformers, would recover the majority of that time.

However, we must mention that our implementation relies on an older version of TVLA. Recent developments significantly improved the overall performance of TVLA, and the performance of the coerce operation, in particular [15, 79]. The reported speedups range from $40\times$ to $200\times$.

Another factor that slows down the analysis is our use of the polyhedral numeric domain. While offering superior precision, the polyhedral numeric domain does not scale well as the number of dimensions grows. This property is particularly apparent when a polyhedron that represents the abstract state is a multidimensional hypercube. In the array-copy example, the constraints on the values of elements of both arrays form a 10-dimensional hypercube, which provides an explanation of why the analysis takes over 6 minutes. If the constraints on the values of array a are excluded from the initial abstract state, the analysis takes merely 8 seconds.

Observation of the numeric constraints that arise in the course of the analysis led us to believe that using the less precise, but more efficient weakly-relational domain of zone intervals [86], may speed up the analysis of the above examples without sacrificing precision. We reran the analysis of the array-copy example, using a summarizing extension of this weakly-relational domain. The analysis was able to synthesize the desired property in 40 seconds, which is a significant improvement over the time it takes to perform the analysis with a polyhedral domain.

4.7 Related work

The problem of reasoning about values stored in arrays has been addressed in previous research. Below, we present a comprehensive survey of existing array-analysis techniques.

Masdupuy, in his dissertation [81], uses numeric domains to capture relationships among values and index positions of elements of *statically initialized* arrays. In contrast, our framework allows to discover such relationships for *dynamically initialized* arrays. In particular, canonical abstraction lets our approach retain precision by handling assignments to array elements using strong updates.

Blanchet et al., while building a *special-purpose static program analyzer* [14], recognized the need for handling values of array elements. They proposed two practical approaches: (i) *array expansion*, i.e., introducing an abstract element for each index in the array; and (ii) *array smashing*, i.e., using a single abstract element to represent all array elements. Array expansion is precise, but in practice can only be used for arrays of small size, and is not able to handle unbounded-sized arrays. Array smashing allows handling arbitrary arrays efficiently, but suffers precision losses due to the need to perform weak updates. Our approach combines the benefits of both array expansion and array smashing by dynamically expanding the elements that are read or written so as to avoid weak updates, and smashing together the remaining elements.

Flanagan and Qadeer used predicate abstraction to infer universally-quantified loop invariants [43]. To handle unbounded arrays, they used special predicates over *Skolem constants*, which are synthetically introduced variables with unconstrained values. These variables are then quantified out from the inferred invariants. The predicates are either supplied manually, or derived from the program code with some simple heuristics. Our approach is different in that we model the values of all array elements directly and use *summarization* to handle unbounded arrays. Also, our approach uses abstract numeric domains to maintain the numeric state of the program, which obviates the need for calls to a theorem prover and for performing abstraction refinement.

Lahiri et al., [73] proposed a heuristic for deriving *indexed* predicates (similar to the predicates in Flanagan and Qadeer’s technique) that can be used by predicate abstraction to infer universal array properties. The heuristic computes the weakest liberal precondition for the predicates that appear in the property to be verified, and uses a set of simple rules to select certain conjuncts from the weakest-precondition formula. In contrast, our approach models the values of array elements directly, and thus, in most cases, does not require special predicates to specify the properties of array elements. However, in cases when our technique requires auxiliary predicates, either due to summarization (as in the insertion-sort example) or due to logical implications in the universal property to be verified (as in array copy), predicate abstraction (seeded with the right set of predicates) may have an edge over our technique.

Černý introduced the technique of *parametric predicate abstraction* [111], in which special-purpose abstract domains are designed to reason about properties of array elements. The domains are parametrized with numeric quantities that designate the portion of an array for which the desired property holds. The abstract transformers are manually defined for each domain. The analysis instantiates the domains by explicitly modeling their parameters in the numeric state. Our approach differs in two respects. First, in our approach numeric states directly model array elements, which allows the analysis to automatically synthesize certain invariants that involve values of array elements. Second, our approach separates the task of array partitioning from the task of establishing the properties of array elements. This separation allows the user to concentrate directly on formulating auxiliary predicates that capture the properties of interest.

Armando et al. proposed a technique for model checking linear programs with arrays [2–4]. The technique is based on abstraction refinement in the sense that the set of variables and array elements modeled by the analysis may be refined during analysis: i.e., the technique starts by modeling only scalar variables. If the property cannot be verified, certain array elements are brought into consideration. The process is repeated iteratively, on each step more and more array elements are tracked by the analysis. The primary difference of this approach from our work is that it is not capable to derive universal properties for unbounded arrays.

Jhala and McMillan proposed a technique for deriving array abstractions from the proofs of infeasibility of spurious counterexamples [67]. The technique uses Craig Interpolation to derive *range* predicates, which state that certain property holds for array elements with indices in a certian symbolic range. These predicates are very similar to the parametric predicates used by Černý [111]. Our approach differs in that we model the values of array elements directly, and thus we do not have to rely on iterative refinement to produce the necessary predicates.

Chapter 5

Guided Static Analysis

The goal of static analysis is as follows: given a program and a set of initial states, compute the set of states that arise during the execution of the program. Due to undecidability of this problem in general, the sets of program states are typically over-approximated by families of sets that both are decidable and can be effectively manipulated by a computer. Such families are referred to as abstractions or abstract domains. As we outlined in Chapter 2, if the abstract domain possesses certain algebraic properties—namely, if the abstract transformers for the domain are monotonic and distribute over join, and if the domain does not contain infinite strictly-increasing chains—then simple iterative techniques yield the the most precise approximation for the set of reachable states.

However, many useful existing abstract domains, especially those for modeling numeric properties, do not possess the above algebraic properties. As a result, standard iterative techniques (augmented with widening, to ensure analysis convergence) tend to lose precision. The precision is lost both due to overly-conservative invariant guesses made by widening, and due to joining together the sets of reachable states along multiple paths. Moreover, the precision losses tend to “snowball” throughout the duration of analysis; e.g., an overly-conservative loop-invariant guess can lead the analysis to consider infeasible execution paths through the body of the loop, which, in turn, can cause the analysis to generate an even more conservative invariant guess on the next loop iteration.

In this chapter, we introduce *guided static analysis*, a general framework that improves the precision of program analysis by guiding its exploration of the program’s state space. The framework controls state-space exploration by applying standard program-analysis techniques to a sequence of *program restrictions*, which are modified versions of the analyzed program. The result of each analysis run is used to derive the next program restriction in the sequence, and also serves as an approximation of a set of initial states for the next analysis run. Note that existing static-analysis techniques are utilized “as is”, making it easy to integrate the framework into existing tools. The framework is instantiated by specifying a procedure for deriving program restrictions.

We present two instantiations of the framework, which target the precision loss due to the application of widening operators. Recall from Chapter 2 that standard analysis techniques are composed of two phases: the *ascending iteration* phase, which relies on widening to over-approximate the solution; followed by the *descending iteration* phase, which refines the overly-conservative solution obtained by the first phase. The rationale behind our approach is that the *refinement* phase will be applied multiple times throughout the analysis: i.e., once for each program restriction. Intuitively, this allows the analysis to refine the intermediate approximations of the solution before the precision loss has a chance to “snowball”. Thus, the splitting of the analysis into multiple sequential phases—which is the primary contribution of our approach—allows the ascending and descending iteration sequences to be interleaved. This is something that cannot be done in the context of the standard approach from Chapter 2 because the interaction between uncontrolled widening and narrowing operators may cause non-convergence. In contrast, with our approach convergence is guaranteed (see §5.3 and §5.6.2).

The first instantiation improves the precision of widening in loops that have multiple phases. This instantiation operates by generating a sequence of program restrictions that gradually introduces individual program phases to the analysis. Individual program phases have *simpler* behaviors than that of the entire program; thus, existing program-analysis techniques are able to obtain a more precise solution for each phase.

The second instantiation addresses the precision of widening in loops where the behavior of each iteration is chosen non-deterministically. Such loops occur naturally in the realm of synchronous systems [46, 57] and can occur in imperative programs if some condition within a loop is abstracted away. This instantiation derives a sequence of program restrictions, each of which enables a single kind of iteration behavior and disables all of the others. In the end, to make the analysis sound, a program restriction with all of the behaviors enabled is analyzed. This strategy allows the analysis to characterize each behavior in isolation, thus obtaining more precise results.

In non-distributive domains, the join operation loses precision. To keep the analysis precise, many techniques propagate sets of abstract values instead of individual values. Various heuristics are used to keep the cardinalities of propagated sets manageable. The main question that these heuristics address is which abstract elements should be joined and which must be kept separate. Guided static analysis is comprised of a sequence of phases, where each phase derives and analyzes a program restriction. The phase boundaries are natural points for separating abstract values: that is, within each phase the analysis may propagate a single abstract value; however, the results of different phases need not be joined together, but may be kept as a set, thus yielding a more precise overall result. In §5.5, we show how to extend the framework to take advantage of such disjunctive partitioning.

In §5.6, we present *lookahead widening*, an implementation technique for the instantiation of guided static analysis framework that addresses widening precision in loops with multiple phases. Lookahead widening folds the sequence of standard analysis runs (i.e., one for each program restriction) into a single run of a standard analysis. To achieve this, the abstract domain used by the analysis is extended to propagate two abstract values. The first value is used to keep the analysis within the current loop phase (i.e., within the current program restriction): this value is used to decide “where to go” at program conditionals and is never widened. The second value is used to compute the solution for the current phase: both widening and narrowing are applied to it. When the second value stabilizes, it is promoted into the first value, thereby allowing the analysis to advance to the next phase.

We refer to the first value as the *main value*, because it contains the overall solution after the analysis converges, and to the second value as *the pilot value*, because it “previews” the behavior of the program along the paths to which the analysis is restricted.[1] The overall technique is called *lookahead widening*, because, from the point of view of the main value, the pilot value determines a suitable extrapolation for it by sampling the analysis future.

[1] The word *pilot* is used in the sense of, e.g., a sitcom pilot in the television industry.

Historically, we introduced the technique of *lookahead widening* first [48]. As it is, lookahead widening is an integral part of all program-analysis tools we have implemented. We present a comprehensive experimental evaluation of lookahead widening in §5.7.1. Recently, we generalized the principles behind lookahead widening into the *guided static analysis* framework [49]. §5.7.2 presents preliminary experimental results for guided static analysis, which we obtained with a prototype implementation.

5.1 Preliminaries

Recall from Chapter 2, that a program is specified by a *control flow graph (CFG)* $G = (V, E)$, where V is a set of program locations, and $E \subseteq V \times V$ is a set of edges that represent the flow of control. A *program state* assigns a value to every variable in the program. We use Σ to denote the set of all possible program states. The function $\Pi_G : E \to (\Sigma \to \Sigma)$ assigns to each edge in the CFG the concrete semantics of the corresponding program statement. The semantics of individual statements is trivially extended to operate on sets of states, i.e., $\Pi_(e)(S) = \{\Pi_G(e)(s) \mid s \in S\}$, where $e \in E$ and $S \subseteq \Sigma$.

Let $\Theta_\triangleright : V \to \wp(\Sigma)$ denote a mapping from program locations to sets of states. The sets of program states that are *reachable* at each program location from the states in Θ_0 are given by the least map $\Theta_\star : V \to \wp(\Sigma)$ that satisfies the following set of equations (Eqn. (2.1)):

$$\Theta_\star(v) \supseteq \Theta_\triangleright(v), \quad \text{and} \quad \Theta_\star(v) = \bigcup_{\langle u,v \rangle \in E} \Pi_G(\langle u,v \rangle)(\Theta_\star(u)), \text{ for all } v \in V$$

The problem of computing exact sets of reachable states is, in general, undecidable.

5.1.1 Static Analysis

Static analysis sidesteps undecidability by using abstraction: sets of program states are approximated by elements of some abstract domain $\mathbb{D} = \langle D, \sqsubseteq, \top, \bot, \sqcup, \nabla \rangle$. Let α and γ specify the abstraction and the concretization functions for the domain $\mathbb{D}$, respectively. The function $\Pi_G^\sharp : E \to (D \to D)$ gives the abstract semantics of individual program statements. To refer to abstract states at multiple program locations, we use *abstract-state maps* $\Theta^\sharp : V \to D$. Similar to Chapter 2, we define the operations α, γ, $\sqsubseteq$, and $\sqcup$ for $\Theta^\sharp$ as pointwise extensions of the corresponding operations for the domain $\mathbb{D}$.

A static analysis computes an approximation for the set of states that are reachable from an approximation of the set of initial states according to the abstract semantics of the program. In the rest of the paper, we view static analysis as a *black box*, denoted by Ω, with the following interface: $\Theta^\sharp_\star = \Omega(\Pi^\sharp_G, \Theta^\sharp_\triangleright)$, where $\Theta^\sharp_\triangleright = \alpha(\Theta_\triangleright)$ is the initial abstract-state map, and $\Theta^\sharp_\star$ is an abstract-state map that satisfies the property from Eqn. (2.2), i.e.,

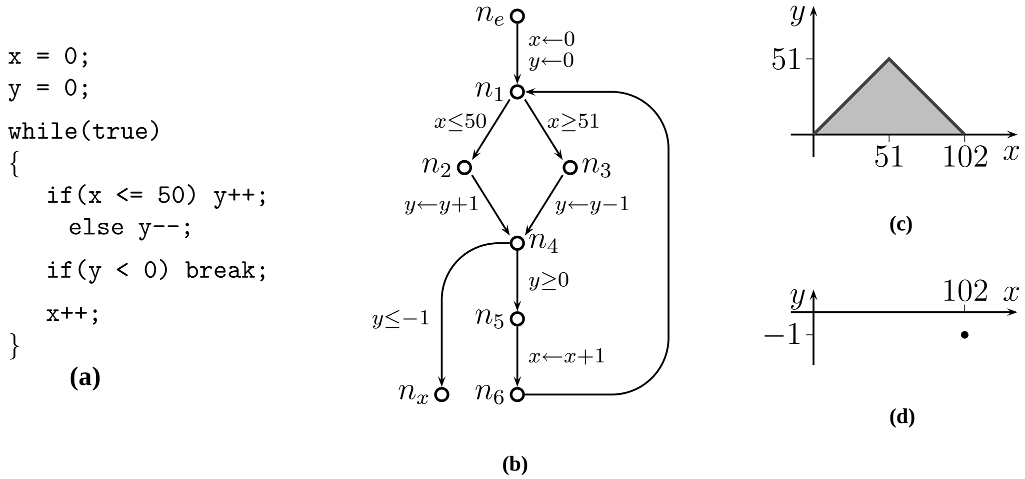
$$\forall v \in V : \left[\Theta^\sharp_\triangleright(v) \sqcup \bigsqcup_{\langle u,v\rangle \in E} \Pi^\sharp{}_G(\langle u, v\rangle)(\Theta^\sharp_\star(u)) \right] \sqsubseteq \Theta^\sharp_\star(v).$$

Chapter 2 demonstrated how to construct a particular static analysis, Ω, based on the abstract-interpretation framework [27, 29].

5.2 Overview of Guided Static Analysis

A *guided-static-analysis* framework provides control over the exploration of the program's state space. Instead of constructing a new analysis by means of designing a new abstract domain or imposing restrictions on existing analyses (e.g., by fixing an iteration strategy), the framework uses an existing static analysis “as is”. Instead, state-space exploration is guided by modifying the analyzed program to restrict some of its behaviors; multiple analysis runs are performed to explore all of the program's behaviors.

The framework is parametrized with a procedure for deriving such program restrictions. The analysis proceeds as follows: the initial abstract-state map, $\Theta^\sharp_\triangleright$, is used to derive the first program restriction; standard static analysis is applied to that program restriction to compute $\Theta^\sharp_1$, which approximates a set of program states reachable from $\Theta^\sharp_\triangleright$. Then, $\Theta^\sharp_1$ is used to derive the second program restriction, which is in turn analyzed by a standard analysis to compute $\Theta^\sharp_2$. This process is repeated until the i-th derived restriction is equivalent to the original program; the final answer is $\Theta^\sharp_i$.



```
x = 0;
y = 0;
while(true)
{
   if(x <= 50) y++;
     else y--;
   if(y < 0) break;
   x++;
}
```

Figure 5.1 Running example: (a) a loop with non-regular behavior; (b) control-flow graph for the program in (a); (c) the set of program states at n_1: the points with integer coordinates that lie on the dark upside-down “v” form the precise set of concrete states; the gray triangle gives the best approximation of that set in the polyhedral domain; (d) the single program state that reaches n_x.

We use the program in Fig. 5.1(a) to illustrate the guided-static-analysis framework. The loop in the program has two explicit phases: during the first fifty iterations, both variable x and variable y are incremented; during the next fifty iterations, variable x is incremented and variable y is decremented. The loop exits when the value of y falls below 0. This program is a challenge for standard widening/narrowing-based numeric analyses because the application of the widening operator over-approximates the behavior of the first phase and thus initiates the analysis of the second phase with overly-conservative initial assumptions. Fig. 5.2 illustrates the application of standard numeric analysis, using the polyhedral abstract domain, to the program. Widening is performed at node n_1 on the second and third iterations. After the third iteration, an ascending iteration sequence of the analysis converges to a post-fix-point. A descending iteration sequence converges in one iteration: it recovers the precision lost by the application of widening on the third iteration, but is not able to recover the precision lost by the application of widening on the second iteration. As a result, standard numeric analysis concludes that at the program point n_1 the relationship between the values of x and y is $0 \leq y \leq x$, and at the program point n_x, $y = -1$ and

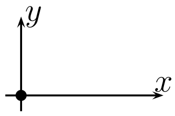
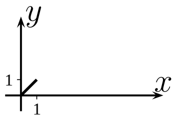
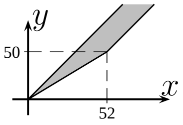
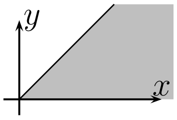
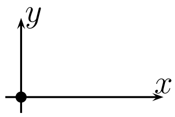
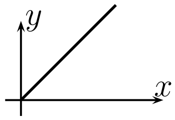
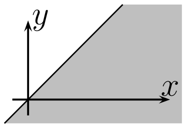
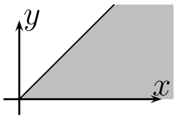
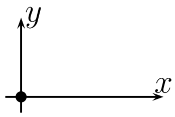
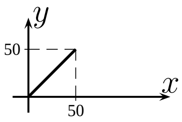
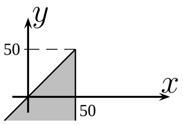
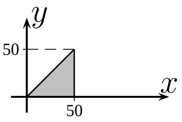
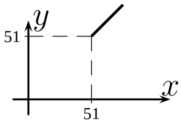
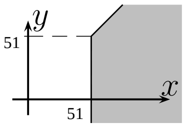
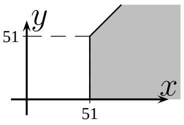
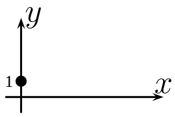
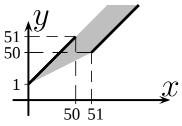
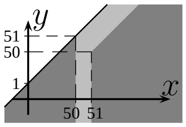
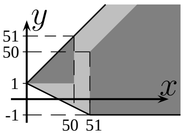
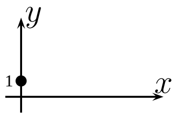
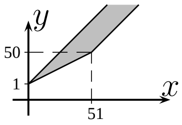
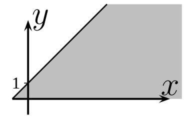
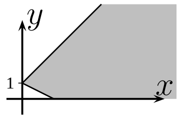
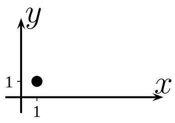
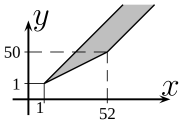
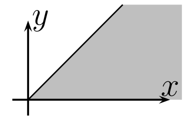
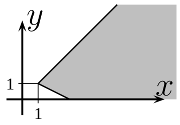
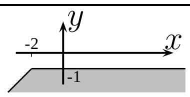
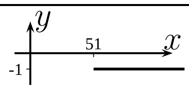
CFG	Ascending iterations			Descending iterations
Node	1st iteration	2nd iteration	3rd iteration	1st iteration
n_e	$\top$	$\top$	$\top$	$\top$
$n_e \sqcup n_6$				
n_1				
n_2				
n_3	$\perp$			
n_4				
n_5				
n_6				
n_x	$\perp$	$\perp$		

Figure 5.2 Standard analysis trace. Widening is performed at node n_1. At the join point, n_4, the polyhedra that are joined are shown in dark gray and the result is shown in light gray.

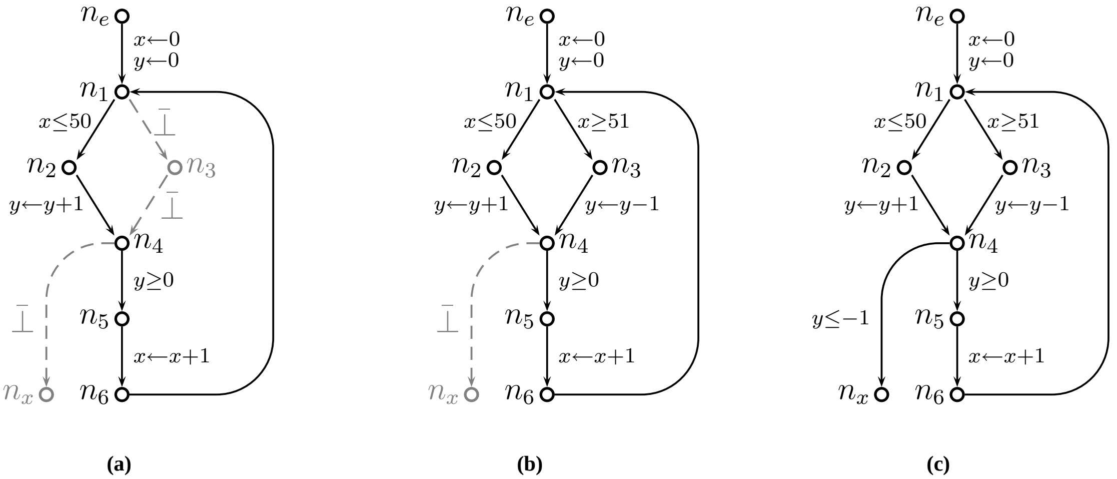


Figure 5.3 Program restrictions for the program in Fig. 5.1: the unreachable portions of each CFG are shown in gray; (a) the first restriction corresponds to the first loop phase; (b) the second restriction consists of both loop phases, but not the loop-exit edge; (c) the third restriction incorporates the entire program.

$x \geq 50$. This is imprecise compared to the true sets of states at those program points (see Figs. 5.1(c) and 5.1(d)).

Guided static analysis, when applied to the program in Fig. 5.1(a) consecutively derives three program restrictions, which are shown in Fig. 5.3: (a) consists to the first phase of the program; (b) incorporates both phases, but excludes the edge that leads out of the loop; (c) includes the entire program. Each restriction is formed by substituting abstract transformers associated with certain edges in the control flow graph with more restrictive transformers (in this case, with $\bar{\bot}$, which is equivalent to removing the edge from the graph). We defer the description of the procedure for deriving these restrictions to §5.4.1.

Fig. 5.4(a) illustrates the operation of guided static analysis. $\Theta_0^\sharp = \Theta_\triangleright^\sharp$ approximates the set of initial states of the program. The standard numeric analysis, when applied to the first restriction (Fig. 5.3(a)), yields the abstract-state map $\Theta_1^\sharp$, i.e., $\Theta_1^\sharp = \Omega(\Pi_1^\sharp, \Theta_\triangleright^\sharp)$. Note that the invariant for the first loop phase ($0 \leq x = y \leq 51$) is captured precisely. Similarly, $\Theta_2^\sharp$ is computed as $\Omega(\Pi_2^\sharp, \Theta_1^\sharp)$, and $\Theta_3^\sharp$ is computed as $\Omega(\Pi_3^\sharp, \Theta_2^\sharp)$. Because the third restriction is equivalent to the program itself, the analysis stops, yielding $\Theta_3^\sharp$ as the overall result. Note that $\Theta_3^\sharp$ is more precise than the solution

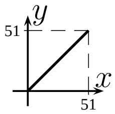
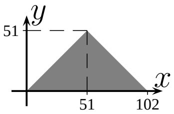
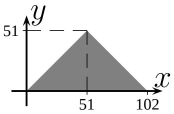
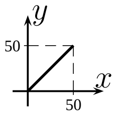
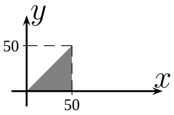
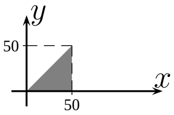
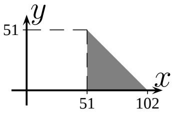
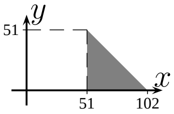
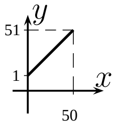
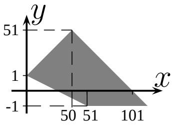
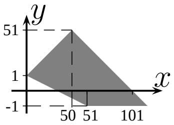
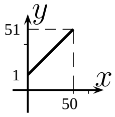
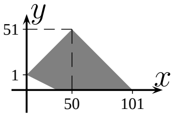
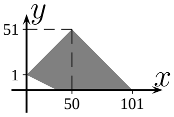
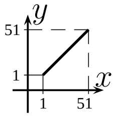
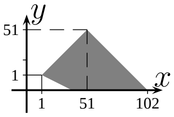
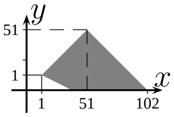
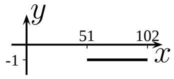
Node	$\Theta_0^\sharp$	$\Theta_1^\sharp$	$\Theta_2^\sharp$	$\Theta_3^\sharp$
n_e	$\top$	$\top$	$\top$	$\top$
n_1	$\bot$			
n_2	$\bot$			
n_3	$\bot$	$\bot$		
n_4	$\bot$			
n_5	$\bot$			
n_6	$\bot$			
n_x	$\bot$	$\bot$	$\bot$	

Figure 5.4 Guided static analysis results for the program in Fig. 5.1(a): the sequence of abstract states that are computed by analyzing the program restrictions shown in Fig. 5.3; abstract-state map $\Theta_3^\sharp$ is the overall result of the analysis.

computed by the standard analysis: it precisely captures the loop invariant at program point n_1 and the upper bound for the value of x at node n_x. In fact, $\Theta_3^\sharp$ corresponds to the least abstract state map that satisfies Eqn. (2.2) (i.e., the least fix-point) for the program in Fig. 5.1(a) in the polyhedral domain.

5.3 Guided Static Analysis

We now define guided static analysis formally. We start by extending the partial order of the abstract domain to abstract transformers and to entire programs. The order is extended in a straightforward fashion.

Definition 5.1 (Program Order) Let $f, g : D \to D$ be two abstract transformers, let $G = (V, E)$ be a control-flow graph, and let $\Pi_1^\sharp, \Pi_2^\sharp : E \to (D \to D)$ be two programs specified over G. We define the following two relations:

- $f \mathrel{\bar{\sqsubseteq}} g \quad \triangleq \quad \forall d \in D \ [f(d) \sqsubseteq g(d)]$
- $\Pi_1^\sharp \mathrel{\dot{\sqsubseteq}} \Pi_2^\sharp \quad \triangleq \quad \forall e \in E \ \left[\Pi_1^\sharp(e) \bar{\sqsubseteq} \Pi_2^\sharp(e)\right]$.

A program restriction is a version of a program $\Pi^\sharp$ in which some abstract transformers under-approximate ($\bar{\sqsubseteq}$) those of $\Pi^\sharp$. The aim is to make a standard analysis (applied to the restriction) explore only a subset of the reachable states of the original program. Note, however, that, if widening is used by the analyzer, there are no guarantees that the explored state space will be smaller (because widening is not monotonic, in general).

Definition 5.2 (Program Restriction) Let $G = (V, E)$ be a control-flow graph, and $\Pi^\sharp : E \to (D \to D)$ be a program specified over G. We say that $\Pi_r^\sharp : E \to (D \to D)$ is a *restriction* of $\Pi^\sharp$ if $\Pi_r^\sharp \mathrel{\dot{\sqsubseteq}} \Pi^\sharp$

To formalize guided static analysis, we need a notion of a *program transformer*: that is, a procedure Λ that, given a program and an abstract state, derives a corresponding program restriction. We allow a program transformer to maintain internal states, the set of which will be denoted $\mathbb{I}$. We assume that the set $\mathbb{I}$ is defined as part of Λ.

Definition 5.3 (Program transformer) Let $\Pi^\sharp$ be a program, let $\Theta^\sharp : V \to D$ be an arbitrary abstract-state map, and let $I \in \mathbb{I}$ be an internal state of the program transformer. A *program transformer*, Λ, computes a restriction of $\Pi^\sharp$ with respect to $\Theta^\sharp$, and modifies its internal state, i.e.:

$$\Lambda(\Pi^\sharp, I, \Theta^\sharp) \ = \ (\Pi_r^\sharp, I_r), \quad \text{where } \ \Pi_r^\sharp \mathrel{\dot\sqsubseteq} \Pi^\sharp \ \text{ and } \ I_r \in \mathbb{I}.$$

To ensure the soundness and the convergence of the analysis, we require that the program transformer possess the following property: the sequence of program restrictions generated by a non-decreasing chain of abstract states must converge to the original program in finitely many steps.

Definition 5.4 (Chain Property) Let $(\Theta_i^\sharp)$ be a non-decreasing chain, s.t.,

$$\Theta_0^\sharp \sqsubseteq \Theta_1^\sharp \sqsubseteq ... \sqsubseteq \Theta_k^\sharp \sqsubseteq ....$$

Let $(\Pi_i^\sharp)$ be a sequence of program restrictions derived from $(\Theta_i^\sharp)$ as follows:

$$(\Pi_{i+1}^\sharp, I_{i+1}) = \Lambda(\Pi^\sharp, I_i, \Theta_i^\sharp)$$

where I_0 is the initial internal state for Λ. We say that Λ satisfies the *chain property* if there exists a natural number n such that $\Pi_i^\sharp = \Pi^\sharp$, for all $i \geq n$.

The above property is not burdensome: any mechanism for generating program restrictions can be forced to satisfy the property by introducing a threshold and returning the original program after the threshold has been exceeded.

Definition 5.5 (Guided Static Analysis) Let $\Pi^\sharp$ be a program, and let $\Theta_\triangleright^\sharp$ be an initial abstract-state map. Also, let I_0 be an initial internal state for the program transformer Λ. *Guided static analysis* performs the following sequence of iterations:

$$\Theta_0^\sharp = \Theta_\triangleright^\sharp \qquad \text{and} \qquad \Theta_{i+1}^\sharp = \Omega(\Pi_{i+1}^\sharp, \Theta_i^\sharp), \ \text{where } (\Pi_{i+1}^\sharp, I_{i+1}) = \Lambda(\Pi^\sharp, I_i, \Theta_i^\sharp),$$

until $\Pi_{i+1}^\sharp = \Pi^\sharp$. The analysis result is $\Theta_\star^\sharp = \Theta_{i+1}^\sharp = \Omega(\Pi_{i+1}^\sharp, \Theta_i^\sharp) = \Omega(\Pi^\sharp, \Theta_i^\sharp)$.

Let us show that if the program transformer satisfies the chain property, the above analysis is sound and converges in a finite number of steps. Both arguments are trivial:

Soundness. Let $\Pi_a^\sharp$ be an arbitrary program and let $\Theta_a^\sharp$ be an arbitrary abstract-state map. Due to the soundness of Ω, the following holds: $\Theta_a^\sharp \sqsubseteq \Omega(\Pi_a^\sharp, \Theta_a^\sharp)$. Let $(\Pi_i^\sharp)$ be a sequence of programs, and let $(\Theta_i^\sharp)$ be a sequence of abstract-state maps computed according to the procedure in Defn. 5.5. Because each $\Theta_i^\sharp$ is computed as $\Omega(\Pi_i^\sharp, \Theta_{i-1}^\sharp)$, clearly, the following relationship holds: $\Theta_0^\sharp \sqsubseteq \Theta_1^\sharp \sqsubseteq ... \sqsubseteq \Theta_k^\sharp \sqsubseteq ....$

Because Λ satisfies the chain property, there exists a number n such that $\Pi_i^\sharp = \Pi^\sharp$ for all $i \geq n$. The result of the analysis is computed as

$$\Theta_\star^\sharp = \Theta_n^\sharp = \Omega(\Pi_n^\sharp, \Theta_{n-1}^\sharp) = \Omega(\Pi^\sharp, \Theta_{n-1}^\sharp)$$

and, since $\Theta_\triangleright^\sharp \sqsubseteq \Theta_0^\sharp \sqsubseteq \Theta_{n-1}^\sharp$ (i.e., the n-th iteration of the analysis computes a set of program states reachable from an over-approximation of the set of initial states, $\Theta_\triangleright^\sharp$), it follows that guided static analysis is sound.

Convergence. Convergence follows trivially from the above discussion: because $\Pi_n^\sharp = \Pi^\sharp$ for some finite number n, guided static analysis converges after n iterations.

5.4 Framework Instantiations

The framework of guided static analysis is instantiated by supplying a suitable program transformer, Λ. This section presents two instantiations that are aimed at recovering precision lost due to the use of widening.

5.4.1 Widening in loops with multiple phases

As was illustrated in §5.2, multiphase loops pose a challenge for standard analysis techniques. The problem is that standard techniques are not able to invoke narrowing after the completion of each phase to refine the analysis results for that phase. Instead, narrowing is invoked at the very end of the analysis when the accumulated precision loss is too great for precision to be recovered.

In this section, we present an instantiation of the guided-static-analysis framework that was illustrated in §5.2. To instantiate the framework, we need to construct a program transformer, Λ_{phase}, that derives program restrictions that isolate individual loop phases (as shown in Fig. 5.3). Intuitively, given an abstract-state map, we would like to include into the generated restriction the edges that are immediately exercised by that abstract state, and exclude the edges that require several loop iterations to become active.

To define the program transformer, we again rely on the application of a standard static analysis to a modified version of the program. Let $\hat{\Pi^\sharp}$ denote the version of $\Pi^\sharp$ from which all backedges have been removed. Note that the program $\hat{\Pi^\sharp}$ is acyclic and thus can be analyzed efficiently and precisely.[2] The program transformer $\Lambda_{phase}(\Pi^\sharp, \Theta^\sharp)$ is defined as follows (no internal states are maintained, so we omit them for brevity):

$$\Pi^\sharp_r(\langle u, v\rangle) = \begin{cases} \Pi^\sharp(\langle u, v\rangle) & \text{if } \Pi^\sharp(\langle u, v\rangle)(\Omega(\hat{\Pi^\sharp}, \Theta^\sharp)(u)) \neq \bot \\ \bar{\bot} & \text{otherwise} \end{cases}$$

In practice, we first analyze the acyclic version of the program: i.e., compute $\hat{\Theta^\sharp} = \Omega(\hat{\Pi^\sharp}, \Theta^\sharp)$. Then, for each edge $\langle u, v\rangle \in E$, we check whether that edge should be included in the program restriction: if the edge is active (that is, if $\Pi^\sharp(\langle u, v\rangle)(\hat{\Theta^\sharp}(u))$ yields a non-bottom value), then the edge is included in the restriction; otherwise, it is omitted.

Fig. 5.5 illustrates this process for the program in Fig. 5.1(a). $\hat{\Pi^\sharp}$ is constructed by removing the edge $\langle n_6, n_1\rangle$ from the program. The first column in Fig. 5.5 shows the result of analyzing $\hat{\Pi^\sharp}$ with $\Theta^\sharp_0$ used as the initial abstract-state map. The transformers associated with the edges $\langle n_1, n_3\rangle$, $\langle n_3, n_4\rangle$, and $\langle n_4, n_x\rangle$ yield $\bot$ when applied to the analysis results. Hence, these edges are excluded from the program restriction $\Pi^\sharp_1$ (see Fig. 5.3(a)). Similarly, the abstract-state map shown in the second column of Fig. 5.5 excludes the edge $\langle n_4, n_x\rangle$ from the restriction $\Pi^\sharp_2$. Finally, all of the edges are active with respect to the abstract-state map shown in the third column. Thus, the program restriction $\Pi^\sharp_3$ is equivalent to the original program.

Note that the program transformer Λ_{phase}, as defined above, does not satisfy the chain property from Defn. 5.4: arbitrary non-decreasing chains of abstract-state maps may not necessarily lead to

[2] We use the word "precisely" in the sense that the analysis need not rely on widening.

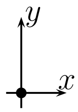
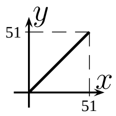
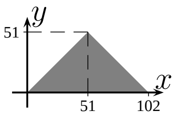
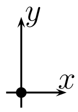
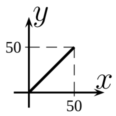
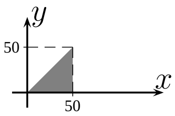
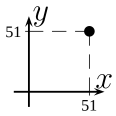
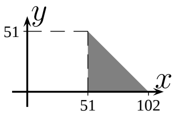
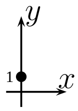
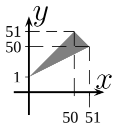
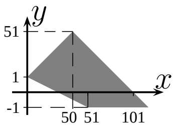
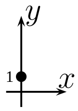
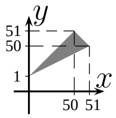
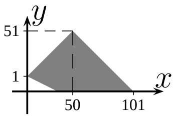
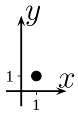
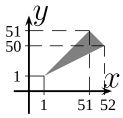
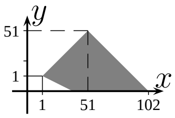
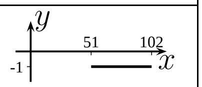
Node	$\Omega(\hat{\Pi}^\sharp, \Theta_0^\sharp)$	$\Omega(\hat{\Pi}^\sharp, \Theta_1^\sharp)$	$\Omega(\hat{\Pi}^\sharp, \Theta_2^\sharp)$
n_e	$\top$	$\top$	$\top$
n_1			
n_2			
n_3	$\bot$		
n_4			
n_5			
n_6			
n_x	$\bot$	$\bot$	

Figure 5.5 The abstract states that are obtained by analyzing the acyclic version of the program in Fig. 5.1(a), which are used to construct the program restrictions in Fig. 5.3 (see §5.4.1).

the derivation of program restrictions that are equivalent to the original program (e.g., unreachable code will not be included in any restriction). However, note that the process is bound to converge to some program restriction after a finite number of steps. To see this, note that each consecutive program restriction contains all of the edges included in the previously generated restrictions, and the overall number of edges in the program's CFG is finite. Thus, to satisfy the chain property, we make Λ_{phase} return $\Pi^\sharp$ after convergence is detected.

5.4.2 Widening in loops with non-deterministically chosen behavior

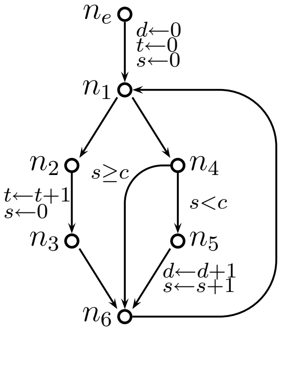
Another challenge for standard analysis techniques is posed by loops in which the behavior of each iteration is chosen non-deterministically. Such loops often arise when modeling and analyzing synchronous systems [46, 57], but they may also arise in the analysis of imperative programs when a condition of an if statement in the body of the loop is abstracted away (e.g., if variables used in the condition are not modeled by the analysis). These loops are problematic due to the following two reasons:

- the analysis may be forced to explore multiple iteration behaviors at the same time (e.g., simultaneously explore multiple arms of a non-deterministic conditional), making it hard for widening to predict the overall behavior of the loop accurately;
- narrowing is not effective in such loops: narrowing operates by filtering an over-approximation of loop behavior through the conditional statements in the body of the loop; in these loops, however, the relevant conditional statements are buried within the arms of a non-deterministic conditional, and the join operation at the point where the arms merge cancels the effect of such filtering.

Fig. 5.6(a) shows an example of such loop: the program models a speedometer with the assumption that the maximum speed is c meters per second ($c > 0$ is an arbitrary integer constant) [46]. Variables m and sec model signals raised by a time sensor and a distance sensor, respectively. Signal sec is raised every time a second elapses: in this case, the time variable t is incremented and the speed variable s is reset. Signal m is raised every time a distance of one meter

```
volatile bool m, sec;
d = t = s = 0;
while(true)
{
   if(sec) {
     t++; s = 0;
   }
   else if(m) {
     if(s < c) {
       d++; s++;
     }
   }
}
```

(a)



(b)

$$0 \leq d = s \leq c$$
$$t = 0$$

(c)

$$s \leq d \leq c \times t + s$$
$$0 \leq d \leq c$$

(d)

$$s \leq d \leq c \times t + s$$
$$0 \leq s \leq c$$

(e)

Figure 5.6 A model of a speedometer with the assumption that maximum speed is c meters per second [46] (c is a positive constant): (a) a program; (b) control-flow graph for the program in (a); (c) abstract state at n_1 after $\Pi_1^\sharp$ (edge $\langle n_1, n_2 \rangle$ disabled) is analyzed; (d) abstract state at n_1 after $\Pi_2^\sharp$ (edge $\langle n_1, n_4 \rangle$ disabled) is analyzed; (e) abstract state at n_1 after $\Pi_3^\sharp = \Pi^\sharp$ is analyzed.

is traveled: in this case, both the distance variable d and the speed variable s are incremented. Fig. 5.6(b) shows the CFG for the program: the environment (i.e., the signals issued by the sensors) is modeled non-deterministically (node n_1). The invariant that we desire to obtain at node n_1 is $d \leq c \times t + s$, i.e., the distance traveled is bound from above by the number of elapsed seconds times the maximum speed, plus the distance traveled during the current second.

Standard polyhedral analysis when applied to this example yields the following sequence of abstract states at node n_1 during the first k iterations (we assume that $k < c$):

$$\{\, 0 \leq s \leq d \leq (k-1) \times t + s,\; t + d \leq k \,\}$$

The application of widening extrapolates the above sequence to $\{\, 0 \leq s \leq d \,\}$ (i.e., by letting k go to ∞). Narrowing refines the result to $\{\, 0 \leq s \leq c,\; s \leq d \,\}$. Thus, unless the widening delay is greater than c, the result obtained with standard analysis is imprecise.

To improve the analysis precision, we would like to analyze each of the loop's behaviors in isolation. That is, we would like to derive a sequence of program restrictions, each of which captures exactly one of the loop's behaviors and suppresses the others. This can be achieved by making each program restriction enable a single edge outgoing from the node where the control is chosen non-deterministically and disable the others. After all single-behavior restrictions are processed, we can ensure that the analysis is sound by analyzing a program restriction where all of the outgoing edges are enabled.

For the program in Fig. 5.6(a), we construct three program restrictions: $\Pi_1^\sharp$ enables edge $\langle n_1, n_4 \rangle$ and disables $\langle n_1, n_2 \rangle$, $\Pi_2^\sharp$ enables edge $\langle n_1, n_2 \rangle$ and disables $\langle n_1, n_4 \rangle$, $\Pi_3^\sharp$ enables both edges. Figs. 5.6(c), 5.6(d), and 5.6(e) show the abstract states $\Theta_1^\sharp(n_1)$, $\Theta_2^\sharp(n_1)$, and $\Theta_3^\sharp(n_1)$ computed by guided static analysis instantiated with the above sequence of program restrictions. Note that the overall result of the analysis in Fig. 5.6(e) implies the desired invariant.

We formalize the above strategy as follows. Let $V_{nd} \subseteq V$ be a set of nodes at which loop behavior is chosen. An internal state of the program transformer keeps track of which outgoing edge is to be enabled next for each node in V_{nd}. One particular scheme for achieving this is to make an internal state I map each node $v \in V_{nd}$ to a non-negative integer: if $I(v)$ is less then the out-degree of v, then $I(v)$-th outgoing edge is to be enabled; otherwise, all outgoing edges are to be enabled. The initial state I_0 maps all nodes in V_{nd} to zero.

If iteration behavior can be chosen at multiple points (e.g., the body of the loop contains a chain of non-deterministic conditionals), the following problem arises: an attempt to isolate all possible loop behaviors may generate exponentially-many program restrictions. In the prototype implementation, we resort to the following heuristic: we simultaneously advance the internal states for *all* reachable nodes in V_{nd}. This strategy ensures that the number of generated program restrictions is linear in $|V_{nd}|$; however, some loop behaviors will not be isolated.

As we illustrate in §5.7.2, the order in which the behaviors are enabled affects the overall precision of the analysis. In our research, we do not address the question of finding an optimal order. In our experiments, we used the above heuristic, and the level of precision achieved is quite good (see §5.7.2). Another possibility would be to randomly choose the order in which behaviors are enabled. To increase the probability of obtaining a precise result, one can perform multiple runs of the analysis and take the meet of the resulting values [105]: each run produces an over-approximation; hence, their meet is also an over-approximation.

Let $deg_{out}(v)$ denote the out-degree of node v; also, let $edge_{out}(v, i)$ denote the i-th edge outgoing from v, where $0 \leq i < deg_{out}(v)$. The program transformer $\Lambda_{nd}(\Pi^\sharp, I, \Theta^\sharp)$ is defined as follows:

$$\Pi_r^\sharp(\langle u, v\rangle) = \begin{cases} \bar{\bot} & \text{if} \left[\begin{array}{c} u \in V_{nd},\ \Theta^\sharp(u) \neq \bot,\ I(u) < deg_{out}(u) \\ \text{and } \langle u, v\rangle \neq edge_{out}(u, I(u)) \end{array} \right] \\ \Pi^\sharp(\langle u, v\rangle) & \text{otherwise} \end{cases}$$

The internal state of Λ_{nd} is updated as follows: for all $v \in V_{nd}$ such that $\Theta^\sharp(v) \neq \bot$, $I_r(v) = I(v) + 1$; for the remaining nodes, $I_r(v) = I(v)$.

As with the first instantiation, the program transformer defined above does not satisfy the chain property. However, the sequence of program restrictions generated according to Defn. 5.4 is bound to stabilize in a finite number of steps. To see this, note that once node $v \in V_{nd}$ becomes reachable, at most $deg_{out}(v) + 1$ program restrictions can be generated before exhausting all of the choices for node v. Thus, we can enforce the chain property by making Λ_{nd} return $\Pi^\sharp$ once the sequence of program restrictions stabilizes.

5.5 Disjunctive Extension

A single iteration of guided static analysis extends the current approximation for the entire set of reachable program states (represented with a single abstract-domain element) with the states that are reachable via the new program behaviors introduced on that iteration. However, if the abstract domain is not distributive, using a single abstract-domain element to represent the entire set of reachable program states may degrade the precision of the analysis. A more precise solution can potentially be obtained if, instead of joining together the contributions of individual iterations, the analysis represents the contribution of each iteration with a separate abstract-domain element.

In this section, we extend guided static analysis to perform such disjunctive partitioning. To isolate a contribution of a single analysis iteration, we add an extra step to the analysis. That step takes the current approximation for the set of reachable program states and constructs an approximation for the set of states that immediately exercise the new program behaviors introduced on that iteration. The resulting approximation is used as a starting point for the standard analysis run performed on that iteration. That is, an iteration of the analysis now consists of three steps: the algorithm (i) derives the (next) program restriction $\Pi_r^\sharp$; (ii) constructs an abstract-state map $\Theta_r^\sharp$ that forces a fix-point computation to explore only the new behaviors introduced in $\Pi_r^\sharp$; and (iii) performs a fix-point computation to analyze $\Pi_r^\sharp$, using $\Theta_r^\sharp$ as the initial abstract-state map.

Definition 5.6 (Analysis History) *Analysis history* H_k captures the sequence of abstract-state maps obtained by the first $k \geq 0$ iterations of *disjunctive* guided static analysis. H_k maps an integer $i \in [0, k]$ to the result of the i-th iteration of the analysis. H_k approximates the set of program states reached by the first k analysis iterations as follows:

$$\gamma(H_k) = \bigcup_{i=0}^{k} \gamma(H_k(i)).$$

The introduction of the analysis history necessitates a change in the definition of a program transformer Λ (Defn. 5.3): instead of a single abstract domain element, a program transformer must accept an analysis history as input. We leave it in the hands of the user to supply a suitable program transformer Λ_{dj}. In our implementation, we used a simple, albeit conservative way to construct such a program transformer from Λ:

$$\Lambda_{dj}(\Pi^\sharp, I, H_k) = \Lambda(\Pi^\sharp, I, \bigsqcup_{i=1}^{k} H_k(i)).$$

For the program in Fig. 5.1, Λ_{dj} derives the same program restrictions as the ones derived by plain guided static analysis (see Fig. 5.3).

Definition 5.7 (Fronier Edges) Let $\Pi_k^\sharp$ be the program restriction derived on the k-th iteration of the analysis, where $k \geq 1$. The set of *frontier edges* for the k-th iteration consists of the edges whose associated transformers are changed in $\Pi_k^\sharp$ from $\Pi_{k-1}^\sharp$ (for convenience, we define $\Pi_0^\sharp$ to map all edges to $\bar{\bot}$):

$$F_k = \left\{ e \in E \mid \Pi_k^\sharp(e) \neq \Pi_{k-1}^\sharp(e) \right\}.$$

For the program in Fig. 5.1, the sets of frontier edges on the second and third iterations are $F_1 = \{\langle n_e, n_1 \rangle, \langle n_1, n_2 \rangle, \langle n_2, n_4 \rangle, \langle n_4, n_5 \rangle, \langle n_5, n_6 \rangle, \langle n_6, n_1 \rangle\}$, $F_2 = \{\langle n_1, n_3 \rangle, \langle n_3, n_4 \rangle\}$ and $F_3 = \{\langle n_4, n_x \rangle\}$.

Definition 5.8 (Local Analysis Frontier) The *local analysis frontier* for the k-th iteration of the analysis is an abstract-state map that approximates the set of states that are immediately reachable via the frontier edges in F_k:

$$\mathit{LF}_k(v) = \bigsqcup_{\langle u,v\rangle \in F_k} \left[\bigsqcup_{i=0}^{k-1} \Pi_k^\sharp(\langle u,v\rangle)(H_{k-1}(i)(u)) \right].$$

For the program in Fig. 5.1, the local analysis frontier on the second iteration contains a single program state: $\mathit{LF}_2(n_3) = \{x = y = 51\}$, which is obtained by applying the transformer associated with the edge $\langle n_1, n_3\rangle$ to the abstract state $H_1(1)(n_1) = \{0 \le x = y \le 51\}$.

Some program states in the local analysis frontier may have already been explored on previous iterations of the analysis. The *global analysis frontier* refines the local frontier by taking the analysis history into consideration.

Definition 5.9 (Global Analysis Frontier) *Global analysis frontier* for the k-th iteration of the analysis is an abstract state map that approximates the set of states in the local analysis frontier LF_k that has not yet been explored by the analysis:

$$\mathit{GF}_k(v) = \alpha(\gamma(\mathit{LF}_k(v)) - \bigcup_{i=0}^{k-1} \gamma(H_{k-1}(i)(v))),$$

where "$-$" denotes set difference.

However, this definition of global analysis frontier is hard to compute in practice. In our implementation, we take a simplistic approach and compute:

$$\mathit{GF}_k(v) = \begin{cases} \bot & \text{if } \mathit{LF}_k(v) \in \{H_{k-1}(i)(v) \mid 0 \le i \le k-1\} \\ \mathit{LF}_k(v) & \text{otherwise} \end{cases}$$

For the program in Fig. 5.1, $\mathit{GF}_2 = \mathit{LF}_2$ and $\mathit{GF}_3 = \mathit{LF}_3$.

Definition 5.10 (Disjunctive Extension) Let $\Pi^\sharp$ be a program, and let $\Theta^\sharp_\triangleright$ be an abstract state that approximates the initial configuration of the program. Also, let I_0 be an initial internal state for the program transformer, Λ_{dj}. The disjunctive extension of guided static analysis computes the set of reachable states by performing the following iteration,

$$H_0 = \left[0 \mapsto \Theta^\sharp_\triangleright\right] \quad \text{and} \quad H_{i+1} = H_i \cup \left[(i+1) \mapsto \Omega(\Pi^\sharp_{i+1}, GF_{i+1})\right],$$
$$\text{where } (\Pi^\sharp_{i+1}, I_{i+1}) = \Lambda_{dj}(\Pi^\sharp, I_i, H_i),$$

until $\Pi^\sharp_{i+1} = \Pi^\sharp$. The result of the analysis is given by H_{i+1}.

Fig. 5.7 illustrates the application of the disjunctive extension to the program in Fig. 5.1(a). The analysis precisely captures the behavior of both loop phases. Also, the abstract value computed for program point n_x exactly identifies the set of program states reachable at n_x.

5.6 Lookahead Widening

The guided-static-analysis framework has extra computational costs associated with it: that is, guided static analysis must perform certain auxiliary operations, which are external to the actual computation of the set of reachable states. For instance, on each iteration of the analysis, the program transformer Λ must be executed. In the case of disjunctive extension, the number of auxiliary operations performed on each iteration is even greater: the operations that have to be executed in addition to Λ include the computation of the local and global analysis frontiers.

In this section, we propose an implementation for the instantiation of guided static analysis in §5.4.1, which targets widening precision in loops with multiple phases. The implementation allows to strip away completely the cost of auxiliary operations for the non-disjunctive guided static analysis: that is, the implementation performs only “useful” work from the point of view of state-space exploration. However, there is certain price to pay: namely, some restrictions must be imposed on the implementation of a standard analysis for it to be usable in the framework; also, the disjunctive extension does not fit well with this implementation scheme.

5.6.1 Approximation of Loop Phases

Instead of deriving syntactic program restrictions explicitly, as we described in §5.3, lookahead widening approximates this behavior by using a specially designed abstract value to guide the

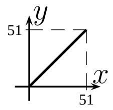
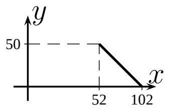
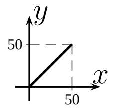
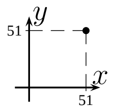
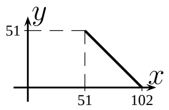
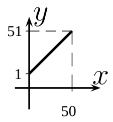
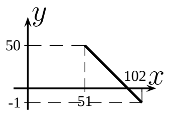
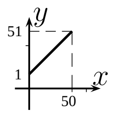
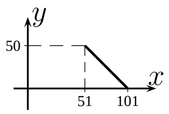
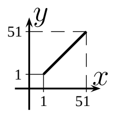
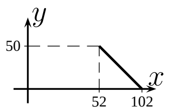
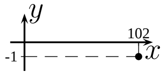
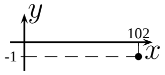
Node	$GF_1 = \Theta_0^\sharp$	$\Theta_1^\sharp$	GF_2	$\Theta_2^\sharp$	GF_3	$\Theta_3^\sharp$
n_e	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$
n_1	$\bot$		$\bot$		$\bot$	$\bot$
n_2	$\bot$		$\bot$	$\bot$	$\bot$	$\bot$
n_3	$\bot$	$\bot$			$\bot$	$\bot$
n_4	$\bot$		$\bot$		$\bot$	$\bot$
n_5	$\bot$		$\bot$		$\bot$	$\bot$
n_6	$\bot$		$\bot$		$\bot$	$\bot$
n_x	$\bot$	$\bot$	$\bot$	$\bot$		

Figure 5.7 Disjunctive extension of guided static analysis: the analysis trace for the program in Fig. 5.1(a); for each analysis phase, the global frontier and the resulting abstract state are shown. Note that the set of abstract values computed for program point n_x describes the true set of states reachable at n_x (see Fig. 5.1(d)).

analysis through the program. That is, the analysis propagates a pair of abstract values: the first value (referred to as *the main value*) is used to decide at conditional points which paths are to be explored; the second value (referred to as *the pilot value*) is used to compute the solution along those paths. Widening and narrowing are only ever applied to the pilot value. Intuitively, the main value restricts the analysis to a particular loop phase, while the pilot value computes the solution for it. After the pilot value stabilizes, it is used to update the main value, essentially switching the analysis to the next syntactic restriction in the sequence.

Let $\mathbb{D}$ be an arbitrary abstract domain: $\mathbb{D} = \langle D, \sqsubseteq, \sqcup, \top, \bot, \nabla, \Delta, \{\tau\}\rangle$, where D is a set of domain elements; $\sqsubseteq$ is a partial order on D; $\sqcup$, $\top$, and $\bot$ denote least-upper-bound operation, the greatest element, and the least element of D with respect to $\sqsubseteq$; ∇ and Δ are the widening operator and the narrowing operator; and $\{\tau : D \to D\}$ is the set of (monotonic) abstract transformers associated with the edges of program's CFG (i.e., the transformers in $\Pi_G^\sharp$—we use τ here to simplify the notation). We construct a new abstract domain:

$$\mathbb{D}_{LA} = \langle D_{LA}, \sqsubseteq_{LA}, \sqcup_{LA}, \top_{LA}, \bot_{LA}, \nabla_{LA}, \{\tau_{LA}\}\rangle\ ,$$

each element of which is a pair of elements of $\mathbb{D}$: one for the main value and one for the pilot value. The pilot value must either equal the main value or over-approximate it. Also, the main value (and consequently the pilot value) cannot be bottom. We add a special element to represent bottom for the new domain:

$$D_{LA} = \{\langle d_m, d_p\rangle \mid d_m, d_p \in D,\ d_m \sqsubseteq d_p,\ d_m \neq \bot\} \cup \{\bot_{LA}\}\ .$$

The top element for the new domain is defined trivially as $\top_{LA} = \langle \top, \top\rangle$.

Abstract transformers are applied to both elements of the pair. However, to make the main value guide the analysis through the program, if an application of the transformer to the main value yields bottom, we make the entire operation yield bottom:

$$\tau_{LA}(\langle d_m, d_p\rangle) = \begin{cases} \bot_{LA} & \text{if } \tau(d_m) = \bot \\ \langle \tau(d_m), \tau(d_p)\rangle & \text{otherwise} \end{cases}$$

We define the partial order for this domain as lexicographic order on pairs:

$$\langle c_m, c_p\rangle \sqsubseteq_{LA} \langle d_m, d_p\rangle \quad \triangleq \quad (c_m \sqsubset d_m) \vee [(c_m = d_m) \wedge (c_p \sqsubseteq d_p)]\ .$$

This ordering allows us to accommodate a decrease in the pilot value by a strict increase in the main value, giving the overall appearance of an increasing sequence. However, the join operator induced by $\sqsubseteq_{LA}$, when applied to pairs with incomparable main values, sets the pilot value to be equal to the main value in the result. This is not suitable for our technique, because joins at loop heads, where incomparable values are typically combined, would lose all the information accumulated by pilots. Thus, we use an over-approximation of the join operator that is defined as a componentwise join:

$$\langle c_m, c_p \rangle \sqcup_{LA} \langle d_m, d_p \rangle = \langle c_m \sqcup d_m, c_p \sqcup d_p \rangle \, .$$

The definition of the widening operator encompasses the essence of our technique: the main value is left intact, while the pilot value first goes through an ascending phase, then through a descending phase, and is *promoted* into the main value after stabilization. Conceptually, the widening operator is defined as follows:

$$\langle c_m, c_p \rangle \, \nabla_{LA} \, \langle d_m, d_p \rangle = \begin{cases} \langle c_m \sqcup d_m, c_p \, \nabla \, d_p \rangle & \text{if the pilot value is ascending} \\ \langle c_m \sqcup d_m, c_p \, \Delta \, d_p \rangle & \text{if the pilot value is descending} \\ \langle d_p, d_p \rangle & \text{if the pilot value has stabilized} \end{cases}$$

The direct implementation of the above definition requires an analyzer to be modified to detect whether the pilot value is in ascending mode, descending mode, or whether it has stabilized. Also, for short phases, there is a possibility that the main value exits the phase before the pilot value stabilizes, in which case the pilot must be switched to ascending mode. These are global properties, and the modifications that are required depend heavily on the implementation of the analyzer. In our implementation, we took a somewhat different route, which we describe in the next section.

5.6.2 Practical Implementation

To simplify the integration of our technique into an existing analyzer, we impose on both the analyzer and the underlying abstract domain restrictions that allow us to check locally the global properties that are necessary for defining a widening operator:

- **R1. Analyzer restriction:** the analyzer must follow a *recursive iteration strategy* [18]; that is, the analysis must stay within each WTO component until the values within that component stabilize. (See §2.3.4 for the definition of recursive iteration strategy.)
- **R2. Abstract domain restriction:** the abstract domain must possess a *stable widening operator* [18]; that is, $x \sqsubseteq y$ must imply that $y \nabla x = y$.

Furthermore, our implementation does not utilize narrowing operators, and only computes the equivalent of a single descending iteration for each loop phase. We believe that this simplification is reasonable because meaningful narrowing operators are only defined for a few abstract domains; also, in the experimental evaluation we did not encounter examples that would have significantly benefited from a longer descending-iteration sequences.

We define the widening operator as follows:

$$\langle c_m, c_p \rangle \nabla_{LA} \langle d_m, d_p \rangle = \begin{cases} \langle c_m, c_p \rangle & \text{if } \langle d_m, d_p \rangle \sqsubseteq_{LA} \langle c_m, c_p \rangle \\ \langle d_p, d_p \rangle & \text{if } d_p \sqsubseteq c_p \\ \langle c_m \sqcup d_m, c_p \nabla d_p \rangle & \text{otherwise} \end{cases}$$

The first case ensures that the widening operator is stable. The second case checks whether the pilot value has stabilized, and promotes it into the main value. Note that the pilot value that is promoted is not c_p, but the value d_p, which was obtained from c_p by propagating it through the loop to collect the effect of loop conditionals (i.e., one possibly-descending iteration is performed). The last case incorporates the pilot's ascending sequence: the main values are joined, and the pilot values are widened.

Soundness. It is easy to see that the results obtained with our technique are sound. Consider the operations that are applied to the main values: they precisely mimic the operations that the standard approach applies, except that widening is computed differently. Therefore, because the application of ∇_{LA} never decreases main values and because main values must stabilize for the analysis to terminate, the obtained results are guaranteed to be sound.

Convergence. We would like to show that a standard analyzer that is constructed in accordance with the principles outlined in Chapter 2 and that employs $\mathbb{D}_{LA}$ as an abstract domain converges.

The use of the recursive iteration strategy (R1) allows us to limit our attention to a single WTO component: that is, if we show that the analysis converges for an arbitrary component, then it must converge for the entire program. Let us focus on the head of an arbitrary component: this is where both widening is applied and stabilization is checked.

First, we show that either the pilot value is promoted or the entire component stabilizes after a finite number of iterations. To do this, we rely on the property of the recursive-iteration strategy that the stabilization of a component can be detected by stabilization of the value at its head [18, Theorem 5]. The main value goes through a slow ascending sequence, during which time the analysis is restricted to a subset of the component's body. The pilot goes through an accelerated ascending sequence, which, if the underlying widening operator ∇ is defined correctly, must converge in a finite number of iterations. ∇_{LA} detects stabilization of the pilot's ascending sequence by encountering a first pilot value (d_p) that is less than or equal to the pilot value on the previous iteration (c_p): because the widening operator is stable (R2), application of widening will not change the previous pilot value. Note that c_p is a (post-)fix-point for the restricted component, and d_p is the result of propagating that (post-)fix-point through the same restricted component, and thus, is itself a (post-)fix-point. Two scenarios must now be considered: either the main value has also stabilized (i.e., $d_m \sqsubseteq c_m$), in which case $\langle d_m, d_p \rangle \sqsubseteq_{LA} \langle c_m, c_p \rangle$ and the entire component stabilizes (due to stability of ∇_{LA}); or the main value has not yet stabilized, in which case the (post-)fix-point d_p is promoted into the main value.

Next, we show that only a finite number of promotions can ever occur. The argument is based on the number of edges in the CFG. Depending on whether or not new CFG edges within the component's body are brought into consideration by the promotion of the pilot value into the main value, two scenarios are possible. If no new edges are brought into consideration, then the analysis stabilizes on the next iteration because both main value and pilot value are (post-)fix-points for this component. Alternatively, new CFG edges are taken into consideration. In this case, the process described in the previous paragraph starts anew, eventually leading to the next promotion. Because the body of the component is finite, new edges can only be brought into consideration a finite number of times. Thus, there can only be a finite number of promotions before the analysis of a component converges.

5.6.3 Revisiting the Running Example

We illustrate the technique of lookahead widening by applying it to our running example. Fig. 5.8 shows the trace of abstract operations performed by the analysis. The first iteration is identical to the standard numeric analysis shown in Fig. 5.2. Differences are manifested on the second iteration: the widening operator propagates the unmodified main value, but applies widening to the pilot value. At node n_4, note that the pilot value has been filtered by the conditional on the edge (n_1, n_2). In contrast, in Fig. 5.2, the abstract state at n_4 on the second iteration has an unbounded band running off to the northeast. On the third iteration, the pilot value that reaches node n_1 is smaller than the pilot value stored there on the second iteration. Thus, this pilot value is promoted into the main value. This corresponds to the solution of the first loop phase from Fig. 5.3(a). As the third iteration progresses, the analysis starts exploring new CFG edges that were brought into consideration by the promotion, in essence, analyzing the program restriction from Fig. 5.3(b).

On the fourth iteration, at n_1, the widening operator is applied to the pilot value again. At n_6, note that the pilot value has been filtered through the conditional on the edge (n_4, n_5). On the fifth iteration, the pilot value is promoted again. From here on, the analysis proceeds in the same fashion as the standard analysis would, and converges on the next iteration. The analysis obtains more precise abstract values at all program points, except for n_2, where the value is the same. Also, note that the resulting solution is similar to that obtained with the first instantiation of guided static analysis (see Fig. 5.4).

5.6.4 Discussion

In this section, we discuss several issues that are of relevance to lookahead widening. These issues include interfacing with existing analyses tools and techniques, and certain limitations of lookahead widening.

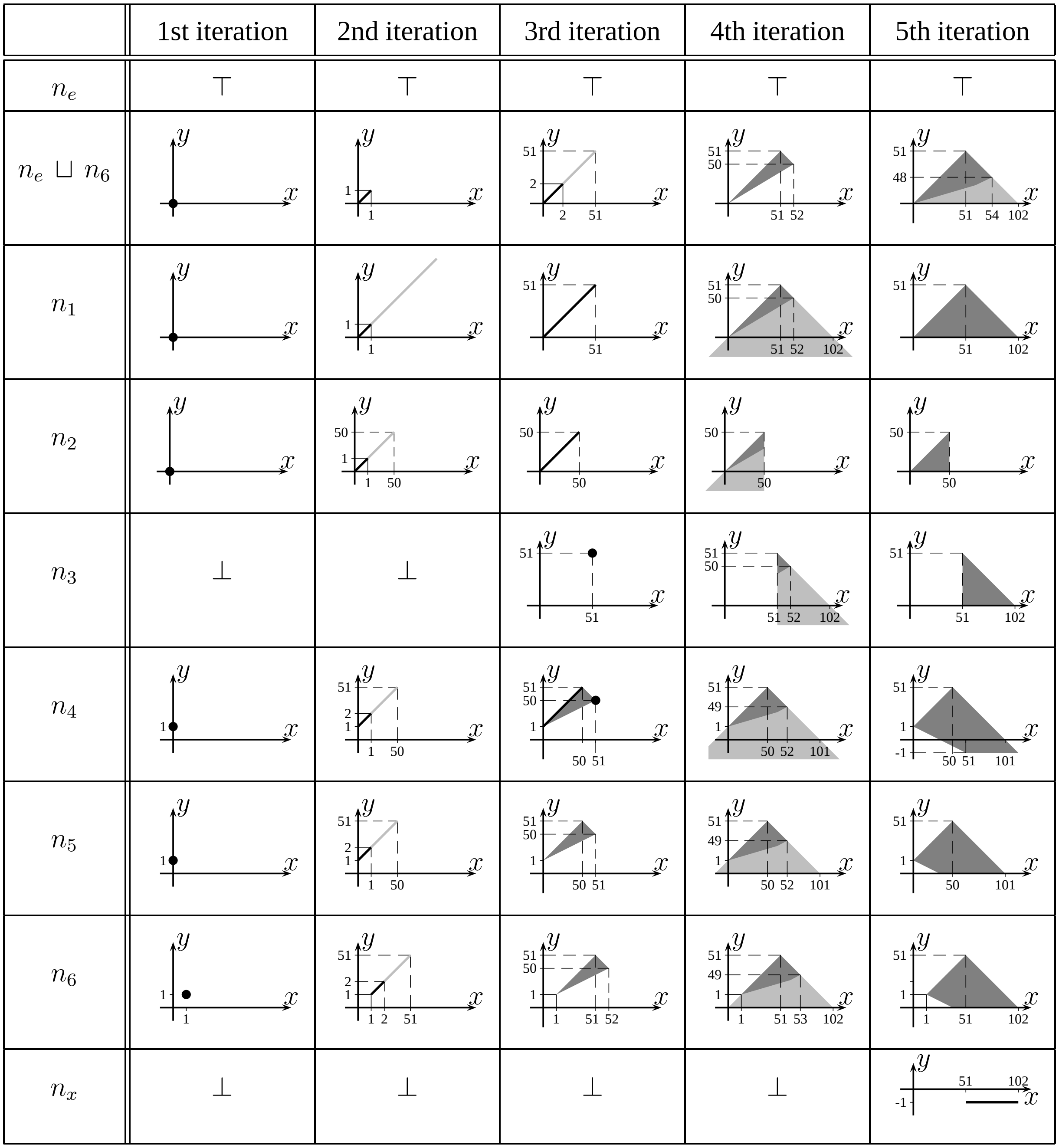


Figure 5.8 Lookahead-widening analysis trace. Widening is applied at node n_1. Main values are shown in dark gray. Light gray indicates the extent of the pilot value beyond the main value. Pilot values are promoted on the 3rd and 5th iterations.

“Accumulating” analyzers. Some analyzers, rather than computing the abstract value for a CFG node $v \in V$ as the join of the values coming from predecessors, i.e., instead of computing

$$\Theta^\sharp_{i+1}(v) = \bigsqcup_{\langle u,v\rangle \in E} \Pi^\sharp(\langle u,v\rangle)(\Theta^\sharp_i(u)),$$

accumulate the abstract value at v by joining the (single) abstract value contributed by a given predecessor u_i to the value stored at v:

$$\Theta^\sharp_{i+1}(v) = \Theta^\sharp_i(v) \ \sqcup\ \Pi^\sharp(\langle m_i, n\rangle))(\Theta^\sharp_i(m_i)).$$

In particular, the WPDS++ implementation of weighted pushdown systems [69], on which our numeric program analysis tool is based, follows this model.

The challenge that such an analyzer design poses to lookahead widening is that the pilot value cannot be promoted directly into the main value by applying ∇_{LA} of the previous section. That is, it is not sound to update v’s value by

$$\Theta^\sharp_{i+1}(v) = \Theta^\sharp_i(v)\ \nabla_{LA}\ \Pi^\sharp(\langle m_i, n\rangle))(\Theta^\sharp_i(m_i))$$

because if the pilot value of $\Pi^\sharp(\langle m_i, n\rangle))(\Theta^\sharp_i(m_i))$ is promoted to be the main value at v, the contributions of other v’s predecessors may be lost.[3] For instance, in Fig. 5.8, on the third iteration, an accumulating analyzer would attempt to widen the value at n_1 with the value at n_6. (The identity transformation is associated with edge $\langle n_6, n_1\rangle$.) The pilot value at n_6 is strictly smaller than the pilot value at n_1, and thus qualifies to be promoted. However, promoting it would result in an unsound main value: the point $(0,0)$ would be excluded.

On the other hand, if the analyzer first performs a join and then widens (as is customarily done in “accumulating” analyzers): i.e.,

$$\Theta^\sharp_{i+1}(v) = \Theta^\sharp_i(v)\ \nabla_{LA}\ \left[\Theta^\sharp_i(v)\ \sqcup_{LA}\ \Pi^\sharp(\langle m_i, n\rangle))(\Theta^\sharp_i(m_i))\right], \tag{5.1}$$

[3] In contrast, in analyzers that update v with the join of the values from all predecessors, any promotion of the pilot in

$$\Theta^\sharp_{i+1}(v) = \Theta^\sharp_{i+1}(v)\ \nabla_{LA} \bigsqcup_{\langle u,v\rangle \in E} \Pi^\sharp(\langle u,v\rangle)(\Theta^\sharp_i(u))$$

does account for the contributions from all predecessors.

then the application of the join operator *cancels* the effects of filtering the pilot value through the conditionals in the body of the loop, thereby reducing lookahead widening into plain widening with a delay.

To allow lookahead widening to be used in such a setting, we slightly redefine the widening operator for accumulating analyzers. In particular, before making decisions about promotion, we join the new pilot value with the main value that is stored at the node. This makes the pilot value account for the values propagated along other incoming edges. The new widening operator is defined as follows:

$$\langle c_m, c_p \rangle \ \nabla_{LA}^{acc} \ \langle d_m, d_p \rangle = \begin{cases} \langle c_m, c_p \rangle & \text{if } \langle d_m, d_p \rangle \sqsubseteq_{LA} \langle c_m, c_p \rangle \\ \langle d_p \sqcup c_m, d_p \sqcup c_m \rangle & \text{if } d_p \sqcup c_m \sqsubseteq c_p \\ \langle d_m \sqcup c_m, c_p \nabla (d_p \sqcup c_m) \rangle & \text{otherwise} \end{cases}$$

Note that for this widening operator to work, the widening must be performed as follows:

$$\Theta_{i+1}^{\sharp}(v) = \Theta_i^{\sharp}(v) \ \nabla_{LA}^{acc} \ \Pi^{\sharp}(\langle m_i, n \rangle))(\Theta_i^{\sharp}(m_i)),$$

and not according to Eqn. (5.1).

Runaway pilots. In loops (or loop phases) that consist of a small number of iterations, it is possible for the analysis to exit the loop (or phase) before the pilot value has stabilized. For instance, if the condition of the if-statement in the running example is changed to $x < 1$, the pilot value will be widened on the second iteration, but will not be effectively filtered through the conditionals because of the contribution from the path through node n_3, which is now enabled by the main value. As a result, the analysis will propagate a pilot value that is larger than desired, which can lead to a loss of precision at future promotions. We refer to this as the problem of *runaway pilots*.

One possible approach to alleviating this problem is to perform a promotion indirectly: that is, instead of replacing the main value with the pilot value, apply widening "up to" [57] to the main values using the symbolic concretization [96] of the pilot value as the set of "up to" constraints. However, we did not try this approach in practice.

Memory usage. The abstract states shown in Fig. 5.8 suggest that the main value and the pilot value are often equal to each other: in our running example, this holds for abstract states that arise on the first, third, and fifth iterations of the analysis (more than half of all abstract states that arise). In our implementation, to improve memory usage, we detect this situation and store a single value instead of a pair of values when the pilot value is equal to the main value.

Delayed widening. Another interesting implementation detail is the interaction of lookahead widening with a commonly used technique called *delayed widening*. The idea behind delayed widening is to avoid applying the widening operator during the first k iterations of the loop, where k is some predefined constant. This allows the abstract states to accumulate more explicit constraints that will be used by the widening operator to generalize the loop behavior. We found it useful in practice to reset the delayed-widening counter after each promotion of the pilot value. Such resetting allows the analysis to perform k widening-free iterations at the beginning of each phase.

5.7 Experimental Evaluation

In this section, we present the experimental evaluation of the techniques that were described in this chapter. We experimented with a stable and time-tested implementation of lookahead widening, and with early prototypes of the two instantiations of guided static analysis.

We compared lookahead widening to standard numeric analysis techniques, which we presented Chapter 2. We built a small analyzer that incorporated both the standard approach and lookahead widening, and applied it to a collection of benchmarks that appeared recently in the literature on widening [14, 26]. Lookahead widening improved analysis precision for half of the benchmarks, with overheads of at most 30% extra analysis iterations (i.e., extra chaotic iterations—see §2.3.4).

Lookahead widening is also a part of our WPDS-based numeric program-analysis tool. There, incorporation of lookahead widening carries special significance: the integration of a descending iteration sequence, which is an integral part of standard numeric analysis, would have required a major redesign of the WPDS++ solver, on which our implementation is based. In contrast, the integration of lookahead widening did not require any modifications to the analysis engine.[4] The integration of lookahead widening allowed our numeric analysis tool to establish tighter loop invariants for 4-40% of the loops in a selected set of benchmarks, with overheads ranging from 3% to 30%.

We implemented prototypes for the instantiations of guided static analysis within the WPDS-based analysis tool. The prototypes were compared against the lookahead-widening implementation on a collection of small benchmarks from [14, 26, 46].

5.7.1 Lookahead-Widening Experiments

We experimented with two implementations of lookahead widening: the first implementation was built into a small intraprocedural analyzer; the second implementation was built into an off-the-shelf weighted-pushdown-system solver, WPDS++ [69]. In both cases, incorporation of lookahead widening required virtually no changes to the analysis engine Both implementations used polyhedral abstract domains built with the Parma Polyhedral Library [7].

Intraprocedural implementation. We applied the first implementation to a number of small benchmarks that appeared in recent papers about widening. The benchmarks `test*` come from work on policy iteration [26]. The `astree*` examples come from [14], where they were used to motivate *threshold widening*: a human-assisted widening technique. `Phase` is our running example, and `merge` is a program that merges two sorted arrays.

Because lookahead widening essentially makes use of one round of descending iteration for each WTO component, we controlled for this effect in our experiments by comparing lookahead widening to a slight modification of the standard widening approach: in Standard+, after each WTO component stabilizes, a single descending iteration is applied to it.[5] This modified analysis converged for all of our benchmarks, and yielded solutions that were at least as precise and often

[4] Weighted pushdown systems, by default, do not support widening. Certain changes had to be made to the engine to make it widening-aware.

[5] In general, interleaving ascending and descending iteration sequences in this way is an unsafe practice and may prevent the analysis from converging.

Program	Vars	Loops	Depth	Standard+		Lookahead		Overhead	Improved
				steps	LFP	steps	LFP	(% steps)	precision (%)
test1	1	1	1	19	yes	19	yes	-	-
test2	2	1	1	24	yes	24	yes	-	-
test3	3	1	1	16	-	19	-	18.8	-
test4	5	5	1	79	-	97	-	22.8	33.3
test5	2	2	2	84	yes	108	yes	28.6	-
test6	2	2	2	110	-	146	-	32.7	100.0
test7	3	3	2	93	no	104	**yes**	11.8	25.0
test8	3	3	3	45	yes	45	yes	-	-
test9	3	3	3	109	yes	142	yes	30.3	-
test10	4	4	3	227	no	266	no	17.2	20.0
astree1	1	1	1	16	no	19	**yes**	18.8	50.0
astree2	1	1	1	27	-	33	-	22.2	-
phase	2	1	1	46	no	58	**yes**	26.1	100.0
merge	3	1	1	63	no	64	**yes**	1.6	100.0

Table 5.1 Lookahead wideining: intraprocedural implementation results. Columns labeled *steps* indicate the number of node visits performed; *LFP* indicates whether the analysis obtains the least-fix-point solution ('-' indicates that we were not able to determine the least fix-point for the benchmark); *improved precision* reports the percentage of *important* program points at which the analysis that used lookahead widening yielded smaller values ('-' indicates no increase in precision). Important program points include loop heads and exit nodes.

more precise than the ones obtained by the standard analysis. The only exception was test10, where the results at some program points were incomparable to the standard technique.

Tab. 5.1 shows the results we obtained. To determine least-fix-points, we ran the analysis without applying widening.[6] The results indicate that lookahead widening achieved higher precision than the strengthened standard approach on half of the benchmarks. Also, the cost of running lookahead widening was not extremely high, peaking at about 33% extra node visits for test6.

We will discuss one benchmark in detail. In astree1, an inequation is used as the loop condition; e.g.,

[6] For some benchmarks, this approach was not able to produce the LFP solution: test3 and astree2 contain loops of the form "while(1) {...}" and so the analysis failed to terminate with widening turned off; programs test4 and test6 terminate, but polyhedral analysis of them (with no widening) does not. This is due to the fact that the set of polyhedra over rational numbers (as implemented in Parma PPL) does not form a complete lattice: i.e., it may contain chains of polyhedra that do not have upper bounds; e.g., consider a sequence of one-dimensional polyhedra (intervals), whose lower bound is fixed and whose upper bounds gradually approach to an irrational number.

Name	Program		Push-down System				Time (sec)		Overhead	Improved
	instr	coverage	stack	same	push	pop	std	look	(%)	precision
		(%)	sym	level				ahead		(%)
speex	22364	7.9	517	483	26	20	1.13	1.33	17.4	40.0
gzip	13166	29.0	1815	2040	76	20	5.70	7.32	28.4	38.2
grep	30376	22.0	9029	10733	201	39	18.62	20.61	10.7	3.3
diff	142959	24.7	9516	11147	217	67	28.41	32.87	15.7	7.5
plot	119910	27.5	15536	15987	1050	159	44.08	45.41	3.0	20.3
graph	129040	26.0	16610	17800	824	155	53.92	56.67	5.1	19.8
calc	178378	18.7	26829	28894	1728	241	85.33	92.23	9.3	5.2

Table 5.2 Lookahead widening: WPDS implementation results. *Instr* lists the number of x86 instructions in the program. *Coverage* indicates what portion of each program was analyzed. *Stack symbols* correspond to program points: there are (roughly) two stack symbols per basic block. *Same-level* rules correspond to intraprocedural CFG edges between basic blocks; *push* rules correspond to procedure calls; *pop* rules correspond to procedure returns. Reported times are for the WPDS *poststar* operation. Precision improvement is given as the percentage of loop heads at which the solution was improved by the lookahead-widening technique.

```
i = 0;
while(i != 100)
   i++;
```

The inequation '$i \neq 100$', which is hard to express in abstract domains that rely on convexity, is modeled by replacing the corresponding CFG edge with two edges: one labeled with '$i < 100$', the other labeled with '$i > 100$'. The application of widening extrapolates the upper bound for `i` to $+\infty$; the descending iterations fail to refine this bound. In contrast, lookahead widening is able to obtain the precise solution: the main value, to which widening is not applied, forces the analysis to always follow the '$i < 100$' edge, and thus the pilot value picks up this constraint before being promoted.

WPDS implementation. We used the WPDS++ implementation to determine linear relations over registers in x86 executables. CodeSurfer/x86 was used to extract a pushdown system from the executable. The contents of memory were not modeled and reads from memory were handled conservatively, e.g., by assigning the value $?$ to the corresponding register (see §2.1.2). Also, we chose to ignore unresolved indirect calls and jumps: as the result, only a portion of each program was analyzed. We applied this implementation to a number of GNU Linux programs that were

Prog.	Lookahead	GSA				Disjunctive GSA			
	steps	phases	steps	prec.	speedup (%)	phases	steps	prec.	speedup (%)
test1	58	2	54	-	**7.9**	2	42	-	**22.2**
test2	56	2	56	-	-	2	42	-	**25.0**
test3	58	1	44	-	**24.1**	1	42	-	**4.5**
test4	210	6	212	-	-1.0	6	154	-	**27.4**
test5	372	3	368	-	**1.1**	3	406	1/3	-10.3
test6	402	3	224	3/3	**44.3**	3	118	2/3	**47.3**
test7	236	3	224	-	**3.4**	3	154	4/4	**31.3**
test8	106	4	146	-	-37.7	3	114	-	**21.9**
test9	430	4	444	-	-3.3	4	488	4/4	-9.9
test10	418	4	420	-	-0.5	4	246	5/5	**41.4**

Table 5.3 Guided static analysis: loops with multiple phases (§5.4.1): GSA is compared against lookahead widening; disjunctive GSA is compared against GSA. *steps* is the total number of steps perfomed by each of the analyses; *phases* is the number of GSA phases; *prec* reports precision improvement: "-" indicates no imrovement, k/m indicates that sharper invariants are obtained at k out of m "interesting" points (interesting points include loop heads and exit nodes);

compiled under Cygwin. The lookahead-widening technique was compared to standard widening. No descending-iteration sequence was applied, because it would have required a major redesign of the WPDS++ solver. Tab. 5.2 presents the results obtained: lookahead widening improves the precision of the analysis on all of the benchmarks, and runs with an overhead of at most 30%.

5.7.2 Guided-Static-Analysis Experiments

We implemented a prototype of the guided-static-analysis framework with both of the instantiations from §5.4.1 and §5.4.2 within the WPDS based numeric program analyzer. As we mentioned in §5.6, there are extra operations that are carried out by guided static analysis, such as deriving program restrictions and computing analysis frontiers. In the WPDS setting, there are additional concerns that have to be addressed. Most notably, some of our WPDS techniques, such as the support for local variables [75], are implemented as weight-wrappers (i.e., a layer on top of an existing weight that also exposes a weight interface). These wrappers must be preserved from iteration to iteration of guided static analysis. In our current implementation, we did not attempt to optimize or even speed up these operations. Instead, our primary concern was the precision of the analysis and the efficiency of actual state-space exploration. Thus, we measure the performance of the analysis in terms of *analysis steps*: each step corresponds to an application of a single abstract transformer.[7] Note that, although, guided static analysis seems to sometimes outperform lookahead widening in terms of analysis steps, in practice, guided-static-analysis runs take much longer compared to lookahead-widening analysis runs.

A widening delay of 4 was used in all of the experiments. Speedups (overheads) are reported as the percentage of extra steps performed by the baseline analysis (evaluated analysis), respectively.

We applied the instantiation from §5.4.1 to a subset of benchmarks that were used to evaluate the intraprocedural implementation of lookahead widening. Tab. 5.3 shows the results we obtained. With the exception of `test6`, the results from GSA and lookahead widening are comparable: the precision is the same, and the difference in running times can be attributed to implementation choices. This is something we expected, because GSA is a generalization of the lookahead-widening technique. However, GSA yields much better results for `test6`: in `test6`, the loop behavior changes when the induction variable is equal to certain values. The changes in behavior constitute short loop phases, which cause problems for lookahead widening. Also, GSA stabilized in a fewer number of steps because simpler polyhedra arise in the course of the analysis.

Tab. 5.3 also compares the disjunctive extension to plain GSA. Because the analysis performed in each phase of the disjunctive extension does not have to reestablish the invariants obtained on previous phases, the disjunctive extension requires fewer analysis steps for most of the benchmarks. To compare the precision of the two analyses, we joined the analysis history obtained by the disjunctive extension for each program location into a single abstract value: for half of the benchmarks, the resulting abstract values are still significantly more precise than the ones obtained by plain GSA. Most notably, the two loop invariants in `test6` are further sharpened by the disjunctive extension, and the number of analysis steps is further reduced.

The instantiation in §5.4.2 is applied to a set of examples from [14, 46]: `astree` is the (second) example that motivates the use of threshold widening in [14], `speedometer` is the example used

[7] Note that, due to the difference between implementations, the *steps* in Tab. 5.1 and in Tab. 5.3 are quite different: in Tab. 5.1, one step corresponds to applying a corresponding transformer to *each* predecessor of the node, computing the join of the resulting values, and updating the value of that node; in Tab. 5.3, one step corresponds to applying a transformer associated with a *single* CFG edge and updating the value stored at the destination of that edge. This explains the difference in values between the two tables.

Program	Vars	Nodes	ND	Lookahead		GSA				Overhead
				steps	inv.	runs	phases	steps	inv.	(%)
astree	1	7	1(2)	104	no	2	3	107	**yes**	2.9
speedometer	3	8	1(2)	114	no	2	3	207	**yes**	81.6
gas burner	3	8	2(2)	164	no	4	3.5	182.5	3/4	11.3
gas burner II	4	5	1(3)	184	no	6	4	162	4/6	**-12.0**

Table 5.4 Guided static analysis: loops with non-deterministic behavior (§5.4.2): *ND* $k(m)$ gives the amount of non-determinism: $k = |V_{nd}|$ and m is the out-degree for nodes in V_{nd}; *runs* is the number of GSA runs, each run isolates iteration behaviors in different order; *steps* is the total number of analysis steps (for GSA it is the average across all runs); *phases* is the average number of GSA phases; *inv.* indicates whether the desired invariant is obtained (for GSA, k/m indicates that the invariant is obtained on k out of m runs).

in §5.4.2; the two other benchmarks are the models of a leaking gas burner from [46]. The results are shown in Tab. 5.4: guided static analysis was able to establish the desired invariants for all of the examples.

When defining the instantiation, we did not specify an order in which the loop behaviors are to be enabled. An interesting experimental question is whether there is a dependence between the order in which behaviors are enabled and the precision of the analysis. To answer this question, we enumerated all possible orders in which iteration behaviors can be enabled for these examples. Interestingly, the precision of the analysis on the two gas-burner benchmarks does depend on the order in which the behaviors are enabled. However, if the order is picked randomly, guided static analysis has more than 66% chance of succeeding for these benchmarks. An interesting open question is whether there is a more systematic way for ordering loop behaviors to improve the precision of the analysis. Another possibility is to use multiple runs—taking the meet of the results [105]—as a way to boost the chances of succeeding.

5.8 Related Work

In this section, we discuss some of the techniques that are related to guided static analysis and lookahead widening. We consider three groups of techniques: techniques that have some control over state-space exploration, techniques that address widening precision, and techniques that address precision loss due to non-distributivity of abstraction.

5.8.1 Controlled state-space exploration

Bourdoncle discusses the effect of an iteration strategy on the overall efficiency of analysis [18]. Model checking in general, and *lazy abstraction* [59] in particular, perform state-space exploration in a way that avoids performing joins: in lazy abstraction, the CFG of a program is unfolded as a tree, and stabilization is checked by a special *covering* relation. The *directed automated random testing (DART)* technique [45] restricts the analysis to the part of the program that is exercised by a particular test input; the result of the analysis is used to generate inputs that exercise program paths not yet explored. The analysis is carried out dynamically by an instrumented version of the program. Grumberg et al. construct and analyze a sequence of under-approximated models by gradually introducing process interleavings in an effort to speed up the verification of concurrent processes [54]. We believe that the GSA framework is more general than the above approaches. Furthermore, the GSA instantiations presented in this chapter address the precision of widening, which is not addressed by any of the above techniques.

5.8.2 Widening precision

Improving widening operators [6]. One research direction is the design of more precise widening operators—that is, widening operators that are better at capturing the constraints that are present in their arguments. This approach is orthogonal to our technique: lookahead widening would benefit from the availability of more precise (base-domain) widening operators.

Widening "up to" [57] (a.k.a. *limited widening*). In this technique, each widening point is augmented with a fixed set of constraints, M. The value that is obtained from the application of the standard widening operator is further restricted by those constraints from M that are satisfied by both arguments of the widening operator. Given a well-chosen set of constraints, this technique is very powerful. A number of heuristics are available for deriving these constraint sets. In principle, the propagation of the pilot value by our technique can be viewed as an automatic way to collect and propagate such constraints to widening points. Alternatively, whenever such constraint sets are available (e.g., are derived by some external analysis or heuristic), lookahead widening can utilize them by applying widening “up to” to the pilot values. This will be beneficial when lookahead widening is not able to break a loop into simpler phases (for instance, if a loop contains a non-deterministic conditional).

“New-control-path” heuristic [57]. This heuristic addresses imprecision that is due to new loop behaviors that appear on later loop iterations: it detects whether new paths through the loop body were explored by the analysis on its last iteration—in which case the application of widening is delayed (to let captured relationships evolve before widening is applied). While this heuristic handles the introduction of new loop behaviors well, it does not seem to be able to cope with complete changes in loop behavior, e.g., it will not improve the analysis precision for our running example. The lookahead-widening technique can be viewed as an extension of the new-control-path heuristic: not only the application of widening is delayed when the new control paths become available, but also the solution for the already explored control paths is refined by computing a descending iteration sequence.

Policy iteration [26, 44]. Policy-iteration techniques derive a series of program simplifications by changing the semantics of the meet operator: each simplification is analyzed with a dedicated analysis. We believe that our approach is easier to adopt because it relies on existing and well-understood analysis techniques. Furthermore, existing policy-iteration techniques support the interval abstact domain [26], and certain weakly-relational abstract domains [44], but it is not obvious whether it is possible to extend the technique to support fully-relational abstract domains (e.g., polyhedra).

Widening with landmarks [108]. Widening with landmarks collects unsatisfiable inequalities (landmarks) and uses them as oracles to guide fix-point acceleration: i.e., at widening points, a special technique that extends the polyhedron to the closest landmark is used in place of widening. The technique is similar in spirit to lookahead widening in that it also collects certain information about the analysis “future” in the form of landmarks. However, widening with landmarks requires specially designed extrapolation operators, which (at the time of writing) are only available for the polyhedral domain. In contrast, lookahead widening can be used with any existing abstract domain, and is much easier to implement and integrate into existing analysis tools.

Combining widening and loop acceleration [46]. Gonnord et al. combine polyhedral analysis with acceleration techniques: complex loop nests are simplified by "accelerating" some of the loops. The analysis requires a preprocessing step, which (i) computes the transformers for individual loop behaviors; (ii) accelerates the transformers (that is, computes the transitive closure of the transition relation imposed by the transformer — this can only be done if the transformer falls into one of the categories supported by the analysis); and (iii) replaces the loops in the program with the accelerated transformers. After that, a standard numeric analysis is executed on the modified program. The instantiation in §5.4.2 attempts to achieve the same effect, but does not rely on explicit acceleration techniques, and is much simpler to implement in practice.

5.8.3 Powerset extensions

Disjunctive completion [31] improves the precision of an analysis by propagating sets of abstract-domain elements. However, to allow its use in numeric program analysis, widening operators must be lifted to operate on sets of elements [5]. Sankaranarayanan et al. [102] circumvent this problem by propagating single abstract-domain elements through an elaboration of the control-flow graph (constructed on the fly). *ESP* [34], TVLA [78], and the *trace-partitioning framework* [83] structure abstract states as functions from a specially-constructed finite set (e.g., set of FSM states [34], set of valuations of nullary predicates [78], and a set of trace-history descriptors, respectively) into the set of abstract-domain elements: at merge points, only the elements that correspond to the same member of the set are joined. The disjunctive extension in §5.5 differs from these techniques in two aspects: (i) the policy for separating abstract-domain elements is imposed implicitly by the program transformer; and (ii) the base-level static analysis, invoked on each iteration of GSA, always propagates single abstract-domain elements.

Chapter 6

Numeric Program Analysis with Weighted Pushdown Systems

The program-analysis techniques that we described in Chapter 2 target programs that consist of a single procedure. In reality, however, this is rarely the case: a typical program is composed of a large number of functions, which may invoke each other (possibly recursively). The set of variables used by a program is no longer uniform: it consists of a set of *global* variables, which are visible to all of the functions in the program; also, each function has a set of *local* variables, which are only used by that function. The information transferred between functions is passed either through global variables or through the function's formal parameters: in this section, we assume that a function has a set of *input* parameters, whose values are specified at the call site of the function and are used to pass the information to the invoked function, and a set of *output* parameters, which are used to return the computed information back to the point in the calling function just after the call site (for simplicity, we will view the function's return value as an output parameter).

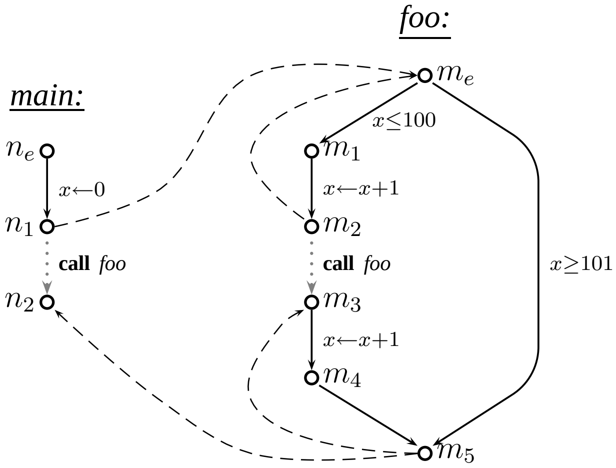
A program is specified as a set of control-flow graphs (CFGs)—one for each function in the program. A special program-state transition, referred to as a *call* transition, is used to invoke a specified function. Also, a subset of nodes of each CFG are designated as *return* nodes: that is, the nodes from which control is transferred back to the calling function.

A straightforward way to apply the techniques from Chapter 2 to perform interprocedural analysis is to connect the CFGs of the individual functions into a single graph, referred to as the *supergraph* [88]. A supergraph is constructed by replacing each call-transition edge in each CFG with a set of edges—an edge from the source of the call-transition edge to the entry node of the

```
main() {;
    x = 0;
    foo();
}

foo() {
    if(x <= 100) {
       x = x + 1;
       foo();
       x = x + 1;
    }
 }
```

(a)



(b)

Figure 6.1 A simple program that consists of two functions: *main* and *foo* (function *foo* is called recursively); (a) textual representation of the program; (b) control-flow graphs for *main* and *foo*; the call-transition edges are shown as dotted arrows; dashed edges illustrate the construction of the supergraph.

called-function's CFG, and an edge from each return node of the called-function's CFG to the destination of the call-transition edge. The entry node of the supergraph corresponds to the entry node of a specially-designated *main* function. The application of the techniques from Chapter 2 to the supergraph of the program yields a sound solution; however, the solution obtained is, in general, not precise.[1]

Fig. 6.1(a) shows a simple program that consists of two functions *main* and *foo*: function *main* calls function *foo*, and function *foo* calls itself recursively. The program has one global variable x. Fig. 6.1(b) shows the control-flow graphs for functions main and *foo*; the supergraph is constructed by replacing the dotted edges in the CFGs with the dashed edges. The program behaves as follows: the first 101 invocations of *foo* follow the left branch of the conditional (the recursive case) bringing the value of x to 101; also, the run-time stack accumulates 101 unfinished invocations of *foo*. On

[1] For an in-depth discussion of the comparative precision of interprocedural analysis vs. intraprocedural analysis, see for instance [94]. We omit the formal discussion of *meet-over-all-paths* solutions vs. *meet-over-all-valid-paths* solutions, because we primarily work with abstract domains that are non-distributive and rely on extrapolation: thus, we generally have no guarantee of obtaining such solutions in either the intraprocedural case or the interprocedural case.

the 102-nd invocation, the right arm of the conditional (the base case) is followed. After that, the run-time stack is unwound by completing the unfinished invocations of *foo*, which brings the value of x to 202. Thus, the value of variable x at the program point n_2 is 202.

The analysis of the supergraph is inherently *not* able to obtain the precise value for the variable x at n_2. The primary challenge for the analysis is the loop formed by nodes m_3, m_4, and m_5: the number of iterations performed by this loop is determined by the configuration of the run-time stack, which is not modeled by the analysis. That is, the analysis fails to separate the valid interprocedural paths (i.e., paths with matching function calls and returns) from other paths. Thus, the analysis assumes that the control at node m_5 is transferred non-deterministically to either m_3 or n_2, and, at best, computes the following over-approximation for the value of x at n_2: $x \geq 101$.

A well-known technique for improving the precision of the analysis described above is the use of *call-strings* [106]: call-strings are finite prefixes of the run-time stack (the run-time stack itself is *unbounded*) that allow to separate the abstract program states that arise in (a finite number of) different contexts. However, to synthesize the property "$x = 202$" at n_2, the length of the call-strings used by the analysis must be at least 101.

To obtain more precise analysis results, a *functional* approach to interprocedural program analysis was proposed [30, 94, 106]. In the functional approach, the analysis computes an abstract summary for each function in the program: each function summary represents how the abstract program state accumulated at the call site of the function is transformed into the abstract program state at the return site of the function. Note that functional program analyses deviate from the ones described in Chapter 2 in the sense that the "unit" of abstraction is no longer a set of concrete states, but rather a *transformation* on a set of concrete states. A suitable summary for function *foo* in the program in Fig. 6.1(a) is:

$$(x_0 \leq 101) \Rightarrow (x_0 + x' = 202),$$

where x_0 denotes the value of variable x at the call site of *foo* and x' denotes the value of x at the return site. The value of x at the call site n_1 is zero (i.e., $x_0 = 0$); thus, the value of x at the return site n_2 must be 202.

Recently, *Weighted Pushdown Systems (WPDSs)* have emerged as a generalization of the functional approaches to interprocedural program analysis [97]. WPDSs use pushdown-system mechanisms to characterize precisely valid interprocedural control-flow paths in the program. Weights are abstractions of the program's transfer functions (i.e., the function summaries in the previous paragraph are encoded as weights). To be used in a WPDS, the domain of weights must satisfy certain algebraic properties, which we describe in detail in §6.2. In this chapter, we show how to construct a weight domain with the use of an existing numeric abstract domain (we use the polyhedral abstract domain to illustrate the process), and discuss the issues that arise in the construction. The construction is based on the *relational program analysis* [30].

6.1 Preliminaries

Let us extend the terminology of Chapter 2 to support multi-procedural programs. Let *Procs* denote the set of all functions in the program. A program is specified by a set of control-flow graphs $\{G_f \mid f \in \mathit{Procs}\}$. For each $G_f = (V_f, E_f)$, $\mathit{Entry}(G_f) \in V_f$ denotes the unique entry node of G_f, and $\mathit{Ret}(G_f) \in \wp(V_f)$ denotes the set of return nodes of G_f.

Let *GVars* denote the set of global variables. Also, for each function f in the program, let LVars_f and PVars_f denote the local variables and the parameters of f, respectively. For function f, we use $\overline{\mathit{in}}_f \in \mathit{PVars}_f^{k_f}$ to denote an ordered vector of input parameters, and $\overline{\mathit{out}}_f \in \mathit{PVars}_f^{m_f}$ to denote an ordered vector of output parameters. Each parameter in PVars_f must appear either in $\overline{\mathit{in}}_f$ or in $\overline{\mathit{out}}_f$, or in both $\overline{\mathit{in}}_f$ and $\overline{\mathit{out}}_f$.

6.1.1 Program States

A program state assigns to each variable its corresponding value. Similarly to Chapter 2, we use functions to map variables to their values. However, unlike Chapter 2, the functions that map variables to their corresponding values form only part of the program state: the other component of the program state is a *run-time* stack.

Valuations of Program Variables. We use Vars_g to denote the set of variables that can be accessed by program function g. This set includes the global variables, the local variables of g, and the parameters of g:

$$\mathit{Vars}_g = \mathit{GVars} \cup \mathit{PVars}_g \cup \mathit{LVars}_g.$$

For a program function g, the valuations of variables are given by functions $S : \mathit{Vars}_g \to \mathbb{V}$. Note that the domains of variable-valuation functions change from one program function to another. We define the following operations for manipulating variable-valuation functions:

- *Operation add_A:* Let $S : W \to \mathbb{V}$ denote an arbitrary function, and let A denote a set, such that $A \cap W = \emptyset$. The operation $[\![add_A]\!](S)$ yields a function $S' : (W \cup A) \to \mathbb{V}$, such that for all $w \in W$, $S'(w) = S(w)$, and for all $a \in A$, the value is chosen non-deterministically from $\mathbb{V}$.

- *Operation $drop_A$:* Let $S : W \to \mathbb{V}$ denote an arbitrary function, and let A denote a set, such that $A \subseteq W$. The operation $[\![drop_A]\!](S)$ yields a function $S' : (W \setminus A) \to \mathbb{V}$, such that for all $w \in W \setminus A$, $S'(w) = S(w)$.

- *Operation merge:* Let $S_1 : W_1 \to \mathbb{V}$ and $S_2 : W_2 \to \mathbb{V}$ denote arbitrary functions, such that $W_1 \cap W_2 = \emptyset$. The operation $[\![merge]\!](S_1, S_2)$ yields a function $S' : (W_1 \cup W_2) \to \mathbb{V}$, such that for all $w \in W_1$, $S'(w) = S_1(w)$, and for all $w \in W_2$, $S'(w) = S_2(w)$.

Run-Time Stack. The other part of the program state is a *run-time stack*, which stores local information about the functions that have been called, but have not yet returned. Each element of the stack is a tuple $V_f \times (W \to \mathbb{V})$, where $W = \mathit{PVars}_f \cup \mathit{LVars}_f$ for some function $f \in \mathit{Procs}$. Intuitively, each stack element stores a node in the control-flow graph of the corresponding function to which the control should be transferred when the callee returns, as well as the values of the local variables and parameters of the function. There is no bound on the size of the stack: in the presence of recursion the stack may grow arbitrarily large.

Program States. A program state is a tuple $\langle T, S \rangle$, where T is a run-time stack and S is a function that maps program variables (accessible to the function that is currently being executed) to their values. Let *main* be the "entry" function of the program. The initial program state (that is, the program state at the node $\mathit{Entry}(G_{\mathit{main}})$) is a tuple consisting of an empty stack and a function

$S_0 : \mathit{Vars}_{\mathit{main}} \to \mathbb{V}$, which maps variables in $\mathit{Vars}_{\mathit{main}}$ to values that are non-deterministically chosen from $\mathbb{V}$.

6.1.2 Concrete Semantics of the Call Transition

Let $g, f \in \mathit{Procs}$ be two program functions, such that g calls f. Let $\langle u, v\rangle \in E_g$ be the control-flow edge to which the call transition $\bar{x} \leftarrow f(\bar{\phi})$ is associated: in a call transition, $\bar{\phi} \in \Phi^{k_f}$ is a vector of expressions that specify the values for the input parameters of f, and $\bar{x} \in \mathit{Vars}_g^{m_f}$ is a vector of variables to which the output parameters of f are to be assigned.

We break the definition of the concrete semantics for the call transition into two parts. The first part handles the invocation of a function: that is, given a program state at the call site (node u), it constructs the program state at the entry node of the callee; the second part handles the return from a function: that is, given a program state at the return node of a function, it constructs a program state at the return site (node v).

Function Invocation. Let $\langle T, S\rangle$ denote the program state at node u. The program state at the entry node of f (i.e., at the node $\mathit{Entry}(G_f)$) is given by $\langle T', S'\rangle$, where

$$T' = \mathit{push}(T,\ \langle v, [\![\mathit{drop}_{\mathit{GVars}}]\!](S)\rangle)$$

and

$$S' = [\![\mathit{drop}_{\mathit{LVars}_g \cup \mathit{PVars}_g}]\!] \circ [\![\overline{\mathit{in}}_f \leftarrow \bar{\phi}]\!] \circ [\![\mathit{add}_{\mathit{LVars}_f \cup \mathit{PVars}_f}]\!])(S).$$

That is, at a function invocation, the caller's local information (i.e., the values of local variables and parameters) are stored on the stack along with the return point v. The state at the entry of the callee is computed by injecting the local variables and parameters of the callee, initializing the parameters, and eliminating the local information of the caller.[2]

[2] In cases, when function f calls itself recursively, the local variables and parameters of f that are added and dropped by the above transformer are two distinct sets of variables: one for the instance of f that performs a call (these are dropped), and one for the instance of f that is being called (these are added). The notation that we use is too weak to distinguish between these two sets of variables.

Function Return. Let $\langle T, S\rangle$ denote the program state at one of the f's return nodes, such that $\mathit{top}(T) = \langle v, S_l\rangle$ The program state at the return site v is given by $\langle T', S'\rangle$, where

$$T' = \mathit{pop}(T) \quad \text{and} \quad S' = ([\![\mathit{drop}_{\mathit{LVars}_f \cup \mathit{PVars}_f}]\!] \circ [\![\bar{x} \leftarrow \overline{\mathit{out}}_f]\!] \circ [\![\mathit{merge}]\!])(S, S_l).$$

That is, the program state at the return site is constructed by merging together the valuation of variables at the return node of the callee with the caller's local information obtained from the stack, assigning the values of output parameters to the corresponding target variables, and eliminating the local information of the callee.

6.2 Overview of Weighted Pushdown Systems

In this section, we give a brief overview of weighted pushdown systems and show how to use them for performing interprocedural program analysis. We start by showing how to use plain (unweighted) pushdown systems to model precisely the control flow of multi-procedural program. Then, we describe the use of weights for representing program-state transformations. We briefly discuss existing techniques for computing the set of reachable WPDS configurations. In the end, we describe the extension of WPDSs for handling local variables.

For an in-depth discussion of pushdown systems, weighted pushdown systems, and extended weighted pushdown systems, we direct the reader to Reps et al. [97] and Lal et al. [75].

6.2.1 Pushdown Systems

Pushdown system (PDSs) are similar to pushdown automata (PDA), but they do not have an input tape. Rather, they represent transition systems for PDA configurations.

Definition 6.1 A *pushdown system* is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is the set of states or control locations, Γ is the set of stack symbols, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is the set of pushdown rules. A *configuration* of $\mathcal{P}$ is a pair $\langle p, u\rangle$ where $p \in P$ and $u \in \Gamma^*$. A *rule* $r \in \Delta$ is written as $\langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$ where $p, p' \in P$, $\gamma \in \Gamma$ and $u \in \Gamma^*$. The rules in Δ define a *transition relation* $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: if $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, then $\langle p, \gamma u'\rangle \Rightarrow_{\mathcal{P}} \langle p', uu'\rangle$ for all $u' \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a set of configurations C, we use $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$ to denote the sets of configurations that are *backward-reachable* and *forward-reachable*, respectively, from the configurations in C under the transition relation $\Rightarrow$.

To simplify the presentation, we restrict PDS rules to have at most two stack symbols on the right-hand side: i.e., for every rule $r \in \Delta$ of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$, the length of u is at most two ($|u| \leq 2$). The restriction does not decrease the expressiveness of pushdown systems: it has been shown that an arbitrary pushdown system can be converted into one that satisfies this restriction [104].

The control flow of a program is modeled as follows. The set of states P contains a single state: $P = \{p\}$. The set of stack locations corresponds to the set of program points: $\Gamma = \bigcup_{f \in \mathit{Procs}} V_f$. The rules in Δ represent the program transitions (i.e., the edges of the control-flow graphs) as follows:

- For each *intraprocedural* edge $\langle u, v \rangle$ in the program (i.e., edges associated with either an assignment or an assume transition), a rule of the form $\langle p, u \rangle \hookrightarrow \langle p, v \rangle$ is added to Δ.

- For each call-transition edge $\langle u, v \rangle$, which represents a call to function $f \in$ *Procs*, a rule of the form $\langle p, u \rangle \hookrightarrow \langle p, \mathit{Entry}(G_f)\ v \rangle$ is added to Δ.

- For each return node $u \in \mathit{Ret}(G_g)$, where $g \in$ *Procs*, a rule of the form $\langle p, u \rangle \hookrightarrow \langle p, \varepsilon \rangle$ is added to Δ.

With this construction, a PDS configuration can be thought of as a CFG node with its calling context, i.e., the stack of return addresses of unfinished calls leading up to the node. The number of possible PDS configurations is unbounded (because the stack can be arbitrarily large). To effectively encode possibly-unbounded sets of configurations finite automata are used.

Definition 6.2 Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system. A *$\mathcal{P}$-automaton* is a finite automaton $(Q, \Gamma, \rightarrow, P, F)$, where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states. A configuration $\langle p, u \rangle$ is represented by a $\mathcal{P}$-automaton if the automaton accepts u (a string over stack alphabet Γ) starting from the initial state p. A set of configurations is called *regular* if there exists a $\mathcal{P}$-automaton that accepts it.

The result that paved the way for model-checking pushdown systems states that for a *regular* set of configurations C the sets $post^*(C)$ and $pre^*(C)$ are also *regular* [16, 39, 41, 104].

6.2.2 Weighted Pushdown Systems

Pushdown systems provide machinery for checking, in a multi-procedural setting, whether a particular program point may be reachable by some execution of the program. However, from the point of view of program analysis, we would like to know not only whether a program point may be reachable, but also the program properties that may arise there. To answer this question, *weighted pushdown systems (WPDSs)* were introduced [17, 97]. WPDSs combine pushdown-system machinery with a domain of weights, which must be a bounded idempotent semiring. Intuitively, pushdown systems are used to model interprocedurally-valid control-flow paths, and weights are used to capture the program properties that arise along those paths.

Definition 6.3 A *bounded idempotent semiring* is a quintuple $(D, \oplus, \otimes, \overline{0}, \overline{1})$, where D is a set, $\overline{0}$ and $\overline{1}$ are elements of D, and $\oplus$ (the combine operation) and $\otimes$ (the extend operation) are binary operators on D, such that

1. $(D, \oplus)$ is a commutative monoid with $\overline{0}$ as its neutral element (i.e., for all $a \in D$, $a \oplus \overline{0} = a$), where $\oplus$ is idempotent (i.e., $a \in D$, $a \oplus a = a$). Also, $(D, \otimes)$ is a monoid with the neutral element $\overline{1}$ (i.e., for all $a \in D$, $a \otimes \overline{1} = a$).
2. $\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$ we have
$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\ .$$
3. $\overline{0}$ is an annihilator with respect to $\otimes$, i.e., for all $a \in D$, $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$.

4. The partial order $\sqsubseteq$, defined as $a \sqsubseteq b \triangleq a \oplus b = a$, does not have infinite descending chains.[3]

For the purpose of program analysis, the weights in D are the abstract transformers, the extend operation $\otimes$ composes abstract transformers (i.e., to construct an abstract transformer for an entire path), the combine operation $\oplus$ joins abstract transformers transformers (i.e., to merge the transformations along multiple paths), $\overline{0}$ is the transformer that maps all abstract states to $\bot$, and $\overline{1}$ is the identity transformer.

Definition 6.4 A *weighted pushdown system* is a triple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system, $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is a bounded idempotent semiring and $f : \Delta \to D$ is a map that assigns a weight to each pushdown rule.

Let $\sigma \in \Delta^*$ be a sequence of rules. Using f, we can associate a value to σ, i.e., if $\sigma = [r_1, \ldots, r_k]$, then we define $v(\sigma) \triangleq f(r_1) \otimes \ldots \otimes f(r_k)$. Let c and c' denote two configurations of $\mathcal{P}$. If σ is a rule sequence that transforms c to c', we say $c \Rightarrow^{\sigma} c'$. We denote the set of all such rule sequences by *paths*(c, c'), i.e.,

$$\textit{paths}(c, c') = \{\sigma \mid c \Rightarrow^{\sigma} c'\}\,.$$

Definition 6.5 Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a WPDS, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $C \subseteq P \times \Gamma^*$ be a regular set of configurations. The *generalized pushdown predecessor (GPP) problem* is to find for each $c \in P \times \Gamma^*$:

- $\delta(c) \triangleq \bigoplus \{v(\sigma) \mid \sigma \in \textit{paths}(c, c'),\ c' \in C\}$;
- a *witness set* of paths: $w(c) \subseteq \bigcup_{c' \in C} \textit{paths}(c, c')$ such that $\bigoplus_{\sigma \in w(c)} v(\sigma) = \delta(c)$.

The *generalized pushdown successor (GPS) problem* is to find for each $c \in P \times \Gamma^*$:

- $\delta(c) \triangleq \bigoplus \{v(\sigma) \mid \sigma \in \textit{paths}(c', c),\ c' \in C\}$;

[3] Traditionally, the literature on weighted pushdown systems uses data-flow methodology (as opposed to the abstract interpretation methodology used in Chapter 2). That is, the partial order on weights has meaning that is opposite to the meaning of the partial order defined in Chapter 2: the smaller the weight, the less precise it is.

- a *witness set* of paths: $w(c) \subseteq \bigcup_{c' \in C} \textit{paths}(c', c)$ such that $\bigoplus_{\sigma \in w(c)} v(\sigma) = \delta(c)$.

There are techniques available for solving the generalized reachability problems [74, 97]. These techniques use finite automata annotated with weights to symbolically represent sets of weighted-pushdown-system configurations: that is, the techniques accept an automaton that represents the initial set of configurations and return an automaton that represents the set of configurations that are either backward-reachable or forward-reachable from the configurations in the initial set. The weight $\delta(c)$ for a particular configuration c is then extracted from the resulting automaton. If the domain of weights is not distributive: that is, if condition 2 in Defn. 6.3 is not satisfied, the existing techniques can still be used, but they yield a conservative approximation for $\delta(c)$.

Definition 6.6 Given a WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, a $\mathcal{W}$*-automaton* $\mathcal{A}$ is a $\mathcal{P}$-automaton, where each transition is labeled with a weight. The weight of a path in the automaton is obtained by computing an extend of the weights on the transitions in the path either in the forward direction (in a *forward* $\mathcal{W}$-automata) or in the backward direction (in a *backward* $\mathcal{W}$-automata). The automaton $\mathcal{A}$ is said to accept a configuration $\langle p, u \rangle$ with weight w if w is the combine of weights of *all* accepting paths for u starting from state p in the automaton.

The techniques for solving the generalized predecessor problem use forward $\mathcal{W}$-automata, whereas the techniques for solving generalized successor problem use backward $\mathcal{W}$-automata.

6.2.3 WPDS in Program Analysis

Let us briefly outline how to use the WPDS machinery described above to analyze program with the concrete semantics shown in §6.1. First, a WPDS $\mathcal{W}$, which represents the program, is constructed: the rules are constructed in the way described in §6.2.1; each rule is annotated with a weight that describes the corresponding program-state transformation (we address weight construction for numeric program analysis in detail in §6.3).

As we stated in Chapter 2, we are interested in computing the set of program states that are reachable by some execution of the program, starting from the set of initial states. A $\mathcal{W}$-automaton is used to encode the set of initial states. In §6.1, we made the assumption that, initially, the control is at the entry node of the *main* function, the run-time stack is empty, and the values of variables are unconstrained. The $\mathcal{W}$-automaton $\mathcal{A}$ that represents such configurations is trivial: it has two states p and a, where p is the initial state and a is the final state, and a transition from p to a, which is labeled by a stack-alphabet symbol corresponding to the node $\mathit{Entry}(G_{\mathit{main}})$, and annotated with the weight $\overline{1}$.

A procedure for solving the generalized successor problem is applied to $\mathcal{W}$ and $\mathcal{A}$ to yield an automaton $\mathcal{B}$, which represents the set of configurations of $\mathcal{W}$ reachable from the configurations in $\mathcal{A}$. The automaton $\mathcal{B}$ is used to compute, for each program point, a weight that represents the abstract-state transformation from the entry to the program to that program point. There exists an efficient procedure, referred to as *path summary*, for computing these weights from $\mathcal{B}$ [97].

The resulting weights approximate the relation between the program states (variable valuations) at the entry of the program, and the program states (variable valuations) at the corresponding program point. Let Σ_f denote the set of program states that may arise in function f (in our case, $\Sigma = \mathit{Vars}_f \rightarrow \mathbb{V}$). Intuitively, a weight computed for a program point in function f can be viewed as a subset of $\Sigma_{\mathit{main}} \times \Sigma_f$. Note that weights do not explicitly model the run-time-stack component of the concrete program states: instead, the stack is modeled by the procedures for solving GPP and GPS, and by the path-summary computation.

The resulting weights can be used in a variety of ways. A weight can be used directly to summarize the transformations performed by the program: in Chapter 7 we use weights directly to construct summary transformers of library functions. A weight can be projected onto its second component (e.g., Σ_f above) to obtain the approximation for the set of states that can arise at the corresponding program point. Alternatively, a weight can be projected onto its first component (e.g., Σ_{main} above) to obtain the approximation for the set of conditions that must be satisfied at the entry of the program for the corresponding program point to be reached. In Chapter 7, we use the latter approach to generate error preconditions for library functions.

6.2.4 Solving the Generalized Successor Problem

In this section, we give a brief description of how the set of reachable configurations is determined. Also, we describe the extension of weighted pushdown systems that provides a convenient way for handling local variables. For an in-depth coverage of these topics, see [75, 97].

The technique for solving GPS proposed by Reps et al. [97] is similar in spirit to the functional program-analysis approach proposed by Sharir and Pnueli [106].[4] The main idea is to compute, for each function in the program, the weight that approximates the transformation that the function performs. To compute such a weight, the technique starts at the function's entry node with the weight $\overline{1}$. The weights for the immediate successors of the entry node are computed by extending the weight $\overline{1}$ with the weights associated with the corresponding CFG edges. In general, the weight for a node v is computed by extending the weight at each predecessor u of v with the weight associated with the edge $\langle u, v\rangle$, and combining the resulting weights. If a call-site of some function is reached, the analysis starts the exploration of that function (unless the weight that approximates the behavior of that function is already available). The weight at the return site is obtained by extending the weight at the call site of the function with the weight computed for the called function.

Lal et al. proposed a more efficient technique for solving GPS [74]. That technique is also based on the premise that a weight for each function is constructed and used to approximate the behavior of the function at each of its call sites.

As we discussed in §6.1, the sets of variables that can be accessed by the program differ from function to function. As a consequence, the extend operations at function call sites have to perform more work than the extend operations at other program locations: in particular, an extend operation at a call site has to merge the information about the caller's local variables (which come from the weight at the call site) with the information about how the global variables are modified by the callee. Lal et al. proposed an extension to weighted pushdown systems that addresses this problem [75]. The extension uses so-called *merge functions* at function call-sites in place of extend

[4] The technique in [97] is expressed as a saturation procedure for the $wpds$-automaton, and, in general, captures more information than the technique in [106]. In particular, the automaton-based representation gives WPDSs the ability to answer stack-qualified queries.

operations. An individual merge function is associated with each call site in the program: thus, merge functions can be specialized to their corresponding call sites. Generally, the extension allows to keep the definition of the extend operation simple and efficient, and also gives an opportunity for an analyzer to have a cleaner design.

6.3 Numeric Program Analysis

In the previous section, we showed how weighted pushdown systems could be used to carry out interprocedural program analysis. The only thing that we have not yet described is how to construct the weights that approximate numeric program-state transitions. In particular, we need to construct weights for assignment transitions, for assume transitions, and for introducing and eliminating local variables and parameters of functions. Also, we need to show how to construct merge functions.

To construct these weights we use ideas from *relational program analysis* [30]: that is, weights capture the relationships between the values of program variables before and after the transformation. More formally, let W_{in} denote the set of variables that are active before the transformation and let W_{out} denote the set of variables that are active after the transformation (we will use subscripts "*in*" and "*out*" to distinguish between the variables from the two sets); the transformation is represented by a set of functions with signature $(W_{\mathit{in}} \cup W_{\mathit{out}}) \rightarrow \mathbb{V}$, such that each function maps the variables in W_{in} to the values that the corresponding variables had before the transformation, and maps the variables in W_{out} to the values that the corresponding variables have after the transformation is applied. For example, consider the assignment transition "$x \leftarrow x + 5$": there is one input and one output variable (both correspond to the program variable x), and the transformation is represented by the set of functions $\{[x_{\mathit{in}} \mapsto \alpha, x_{\mathit{out}} \mapsto \alpha + 5] \mid \alpha \in \mathbb{V}\}$. In many cases, especially for the transformations associated with WPDS rules, the same set of variables is active before and after the transformation. However, for the transformations along paths that cross function boundaries, the sets of input and output variables may be, and in most cases will be, different.

In the following, we express the transformations that are of interest to us as sets of functions. The actual weights are constructed by approximating these sets of functions by elements of some abstract domain. In principle, any numeric abstract domain can be used to approximate the sets of functions described above. However, weakly-relational abstract domains that restrict the number of variables that can appear in a relationship seem to be at disadvantage in this setting: for instance, to represent the transformation "$x \leftarrow x + y$", the abstract domain needs to capture the relationship $x_{out} = x_{in} + y_{in}$, which cannot be represented by the abstract domains that only allow two variables per constraint (e.g., octagons [85] or TVPLI [109]). In the remainder of this section, we will use the domain of polyhedra to illustrate the constructed weights.

Weights $\overline{0}$ and $\overline{1}$. The weight $\overline{0}$ represents the transformation that is given by the empty set of functions. The weight $\overline{1}$ represents the identity transformation. Let W denote the set of variables that are active before and after the identity transformation, i.e., $W_{in} = \{v_{in} \mid v \in W\}$ and $W_{out} = \{v_{out} \mid v \in W\}$; the identity transformation is given by the set of functions:

$$Id \quad \triangleq \quad \{f : (W_{in} \cup W_{out}) \rightarrow \mathbb{V} \ \mid \ \forall v \in W \ [f(v_{in}) = f(v_{out})]\}$$

Technically, the weights $\overline{0}$ and $\overline{1}$ can be constructed in any existing relational numeric abstract domain: $\overline{0}$ can be represented by the $\perp$ element, and $\overline{1}$ only requires constraints of the form $v_{in} = v_{out}$, which all relational and weakly-relational abstract domains are able to represent. However, in general, we would like weights $\overline{0}$ and $\overline{1}$ to be uniform for all functions in the program: this poses a challenge because the sets of active variables change from function to function. Thus, in our implementation, we inject special objects to represent weights $\overline{0}$ and $\overline{1}$ into the domain of weights, and make the combine and extend operations recognize these objects and behave according to Defn. 6.3.

Extend operation: $\otimes$. The extend operation composes two transformations. The constraint imposed by the extend operation on the two transformations is that the set of variables that are active after the first transformation must be equivalent to the set of variables that are active before the second transformation. Let the first transformation be represented by the set of functions $S_1 : (W_{in} \cup W_{ext}) \rightarrow \mathbb{V}$, and let the second transformation be represented by the set of functions $S_2 : (W_{ext} \cup W_{out}) \rightarrow \mathbb{V}$. The resulting transformation is given by the set of functions

$S : (W_{\mathit{in}} \cup W_{\mathit{out}}) \to \mathbb{V}$, which is constructed as follows

$$S = [\![\mathit{drop}_{W_{\mathit{ext}}}]\!] \left([\![\mathit{add}_{W_{\mathit{out}}}]\!](S_1) \cap [\![\mathit{add}_{W_{\mathit{in}}}]\!](S_2)\right).$$

The *add* and *drop* operations in the definition above are the pointwise extensions of the corresponding operations in §6.1.1: i.e., Let $S : W \to \mathbb{V}$ denote an arbitrary function:

$$[\![\mathit{add}_A]\!](S) \quad \triangleq \quad \{f' : (W \cup A) \to \mathbb{V} \ \mid \ \exists f \in S \ \forall w \in W \ [f'(w) = f(w)]\}$$

and

$$[\![\mathit{drop}_A]\!](S) \quad \triangleq \quad \{f' : (W \setminus A) \to \mathbb{V} \ \mid \ \exists f \in S \ \forall w \in W \setminus A \ [f'(w) = f(w)]\}$$

The extend operation can be trivially approximated by a numeric abstract domain: set intersection is approximated by the meet operation ($\sqcap$), the approximations for *add* and *drop* operations are also easy to construct—we discussed those operations in the context of summarizing abstractions (see §3.3).

Combine operation: $\oplus$. The combine operation yields the disjunction of the two transformations: the effect of the resulting transformation is that either of the two input transformations have been applied. In terms of sets of functions, the combine operation yields the union of the sets of functions that represent the input transformations. The combine operation makes sense only if the signatures of the functions in the two input transformations match. At the level of the abstract domain, the combine operation is approximated by the join operation ($\sqcup$).

Assume transition: *assume*(ψ). Suppose that the assume transition *assume*(ψ) appears in program function $f \in \mathit{Procs}$. Let $\psi \in \Psi$ be a k-ary conditional expression, where each variable $w_i \in \mathit{Vars}_f$ for $i = 1..k$. For the assume transition, the sets of variables in the input and output program states are the same: i.e., $W_{\mathit{in}} = \{v_{\mathit{in}} \mid v \in \mathit{Vars}_f\}$ and $W_{\mathit{out}} = \{v_{\mathit{out}} \mid v \in \mathit{Vars}_f\}$. The program-state transformation is represented by the following set of functions:

$$\{\, f : (W_{\mathit{in}} \cup W_{\mathit{out}}) \to \mathbb{V} \ \mid \ \mathit{true} \in [\![\psi_{\mathit{out}}]\!]_{ND}(f) \ \wedge \ \forall v \in \mathit{Vars}_f \, [f(v_{\mathit{in}}) = f(v_{\mathit{out}})] \,\} ,$$

where ψ_{out} is obtained from ψ by renaming every variable w in ψ to w_{out}. The first conjunct makes sure that the program states constructed by the transformation satisfy the condition ψ; the second conjunct makes sure that the assume transition does not modify individual program states.

At the level of the abstract domain, the weight that approximates the assume transition is constructed as follows:

1. construct the top element of a $(2 \times |\mathit{Vars}_f|)$-dimensional abstract domain;

2. for each variable $v \in \mathit{Vars}_f$ establish the constraint $v_{\mathit{in}} = v_{\mathit{out}}$: this can be done by applying either a sequence of assume transitions of the form $[\![\mathit{assume}(v_{\mathit{in}} = v_{\mathit{out}})]\!]^\sharp$ or a sequence of assignment transitions of the form $[\![v_{\mathit{in}} \leftarrow v_{\mathit{out}}]\!]^\sharp$ to the abstract-domain element constructed in step 1;

3. establish the constraint ψ on the output program states: this is done by applying $[\![\mathit{assume}(\psi_{\mathit{out}})]\!]^\sharp$ to the abstract-domain element constructed in step 2.

We denote the resulting weight by $(\!|\mathit{assume}(\psi)|\!)$.

Example 6.7 We use the polyhedral abstract domain to illustrate the construction of weights. The assume transition $\mathit{assume}(x \leq 100)$ associated with the edge $\langle m_e, m_1 \rangle$ in Fig. 6.1 is approximated by the polyhedron that has the following system of constraints:

$$\{x_{\mathit{in}} = x_{\mathit{out}},\ x_{\mathit{out}} \leq 100\}\,.$$

Assignment transition: $\bar{x} \leftarrow \bar{\phi}$. Suppose that the assignment transition $\bar{x} \leftarrow \bar{\phi}$ appears in the program function $f \in \mathit{Procs}$. Let r denote the length of vectors $\bar{x}$ and $\bar{\phi}$. The variables that appear in $\bar{x}$ and $\bar{\phi}$ come from the set Vars_f. Similarly to the assume transition, the sets of variables in the input and output program states for the assignment transition are the same: $W_{\mathit{in}} = \{v_{\mathit{in}} \mid v \in \mathit{Vars}_f\}$ and $W_{\mathit{out}} = \{v_{\mathit{out}} \mid v \in \mathit{Vars}_f\}$. The transformation is represented by the following set of functions:

$$\left\{ f : (W_{\mathit{in}} \cup W_{\mathit{out}}) \to \mathbb{V} \;\middle|\; \left[\begin{array}{ll} f(v_{\mathit{out}}) \in [\![\bar{\phi}_{\mathit{in}}[i]]\!]_{ND}(f) & \text{if } v = \bar{x}[i] \text{ for some } i \in [1,r] \\ f(v_{\mathit{out}}) = f(v_{\mathit{in}}) & \text{otherwise} \end{array} \right] \right\},$$

where $\bar{\phi}_{\mathit{in}}$ is obtained from $\bar{\phi}$ by renaming every variable w in $\bar{\phi}$ to w_{in}.

At the level of the abstract domain, the weight that approximates the assignment transition may be constructed as follows (there are several different ways in which the weight can be constructed):

1. construct the top element of a $(2 \times |\mathit{Vars}_f|)$-dimensional abstract domain;

2. for each variable $v \in \mathit{Vars}_f$ that does not appear in $\bar{x}$, establish the relationships $v_{\mathit{in}} = v_{\mathit{out}}$: this can be done by applying either a sequence of assume transitions of the form $[\![\mathit{assume}(v_{\mathit{in}} = v_{\mathit{out}})]\!]^\sharp$ or a sequence of assignment transitions of the form $[\![v_{\mathit{in}} \leftarrow v_{\mathit{out}}]\!]^\sharp$ to the abstract-domain element constructed in step 1;

3. for each variable $v \in \mathit{Vars}_f$ that does appear in $\bar{x}$, establish the relationship between v_{out} and the variables in the set W_{in}: this can be done by applying the abstract transformer $[\![\bar{x}_{\mathit{out}} \leftarrow \bar{\phi}_{\mathit{in}}]\!]^\sharp$, where $\bar{x}_{\mathit{out}}$ is constructed from $\bar{x}$ by renaming each variable v in $\bar{x}$ to v_{out}, to the abstract-domain element constructed in step 2.

We denote the resulting weight by $(\!|\bar{x} \leftarrow \bar{\phi}|\!)$.

Example 6.8 The weight for the assignment transition $x \leftarrow x+1$ associated with the edge $\langle m_1, m_2 \rangle$ in Fig. 6.1 is given by the polyhedron formed by a single constraint: $\{x_{\mathit{out}} = x_{\mathit{in}} + 1\}$

Adding and Removing Variables. To support local variables and parameter passing, we need to define weights for modifying the set of variables modeled by the analysis. For instance, at the entry to the function, the local variables for that function must be created; and at the return from the function, the local variables must be removed. Let W denote the set of variables in the input state (i.e., $W_{\mathit{in}} = \{v_{\mathit{in}} \mid v \in W\}$), and let A denote the set of variables that have to be added (i.e., $W_{\mathit{out}} = \{v_{\mathit{out}} \mid v \in W \vee v \in A\}$); the transformation for adding variables is expressed as follows:

$$\{f : (W_{\mathit{in}} \cup W_{\mathit{out}}) \to \mathbb{V} \ \mid \ \forall v \in W \ [f(v_{\mathit{out}}) = f(v_{\mathit{in}})]\}\,.$$

For the transformation that removes variables, let W denote the set of variables in the input state (i.e., $W_{\mathit{in}} = \{v_{\mathit{in}} \mid v \in W\}$), and let A denote the set of variables to be removed (i.e., $W_{\mathit{out}} = \{v_{\mathit{out}} \mid v \in W \setminus A\}$); the transformation is expressed as follows:

$$\{f : (W_{\mathit{in}} \cup W_{\mathit{out}}) \to \mathbb{V} \ \mid \ \forall v \in W \setminus A \ [f(v_{\mathit{out}}) = f(v_{\mathit{in}})]\}\,.$$

At the level of the abstract domain, the above transformations can be approximated trivially by elements of the abstract domains with $(2 \times |W| + |A|)$ and $(2 \times |W| - |A|)$ dimensions, respectively. The weight for adding variables (removing variables) is created as follows:

1. construct the top element of a $(2 \times |W| + |A|)$-dimensional abstract domain (a $(2 \times |W| - |A|)$-dimensional abstract domain);

2. for each variable $v \in W$ (for each variable $v \in W \setminus A$), establish the relationship $v_{\mathit{in}} = v_{\mathit{out}}$: this can be done by applying either a sequence of assume transitions of the form $[\![\mathit{assume}(v_{\mathit{in}} = v_{\mathit{out}})]\!]^\sharp$ or a sequence of assignment transitions of the form $[\![v_{\mathit{in}} \leftarrow v_{\mathit{out}}]\!]^\sharp$ to the abstract-domain element constructed in step 1;

We denote the corresponding weights by $(\!|\mathit{add}_A|\!)$ and $(\!|\mathit{drop}_A|\!)$.

Local Variables. In our implementation, local variables are created on entry to called function and are removed before returning. Also, if the function contains call transitions, the local variables are removed before the exploration of the callee is started. More formally, let $f \in \mathit{Procs}$ denote an arbitrary program function. The weights constructed for the WPDS rules that model the control-flow edges whose source node is $\mathit{Entry}(G_f)$ are *pre-extended* with the weight $(\!|\mathit{add}_{\mathit{LVars}_f}|\!)$: e.g., the following weight is created for the edge $\langle m_e, m_1 \rangle$ in Fig. 6.1: $(\!|\mathit{add}_{\mathit{LVars}_{\mathit{foo}}}|\!) \otimes (\!|\mathit{assume}(x \leq 100)|\!)$.

The weights that are constructed for the WPDS pop rules and push rules, which model the return from the function f and the function calls performed by f, respectively, are extended with the weight that removes f's local variables: $(\!|\mathit{drop}_{\mathit{LVars}_f}|\!)$.

Parameter Passing. Let $g, f \in \mathit{Procs}$ be two program functions, such that g calls f. Also, let $\langle u, v \rangle \in E_g$ be the control-flow edge that is associated with the call transition $\bar{x} \leftarrow f(\bar{\phi})$, where the variables in the vector $\bar{x}$ and the variables in the expressions in $\bar{\phi}$ come from the set Vars_g. To deal with the input parameters of g, we generate special weights for the WPDS rules that model the control-flow edges whose destination node is u. Let w denote the weight that approximates the program-state transition associated with such an edge. The corresponding WPDS rule is annotated with the weight that is constructed as follows:

$$w \otimes (\!|\mathit{add}_{\mathit{PVars}_g}|\!) \otimes (\!|\overline{\mathit{in}}_g \leftarrow \bar{\phi}|\!).$$

The above weight first introduces the parameters for the function g, and then initializes them to their respective values.

To deal with the output parameters of function g, we generate special weights for the WPDS rules that model control-flow edges whose source node is v. Let w denote the weight that approximates the program-state transition associated with such edge. The corresponding WPDS rule is annotated with the following weight:

$$(\!|\bar{x} \leftarrow \overline{\mathit{out}}_g|\!) \otimes (\!|\mathit{drop}_{\mathit{PVars}_g}|\!) \otimes w. \tag{6.1}$$

This weight first copies the values of the output parameters to the corresponding target variables, and then removes the parameters of g from consideration.

Merge functions. Let $g, f \in \mathit{Procs}$ be two program functions, such that g calls f. Also, let $\langle u, v\rangle \in E_g$ be the control-flow edge that is associated with the corresponding call transition. The goal of the merge function is to take the weight computed for the call site of function g (i.e., node u) and the weight that approximates the transformation performed by g, and construct the weight for the return site of g (i.e., node v). We denote these weights by w_u, w_g, and w_v, respectively.

The weight computed for the node u has the following set of output variables:

$$W^u_{\mathit{out}} = \mathit{GVars} \cup \mathit{PVars}_f \cup \mathit{LVars}_f \cup \mathit{PVars}_g.$$

The weight that approximates function g, has the following set of input variables:

$$W^g_{\mathit{in}} = \mathit{GVars} \cup \mathit{PVars}_g$$

We extend both the set of input variables W^g_{in} and the set of output variables W^g_{in} of the weight w_g with the variables in $\mathit{PVars}_f \cup \mathit{LVars}_f$. For each such variable $v \in \mathit{PVars}_f \cup \mathit{LVars}_f$, we add an identity constraint $v_{\mathit{in}} = v_{\mathit{out}}$ to w_g. We denote the resulting weight by ${w_g}'$. The weight for the node v is constructed as follows:

$$w_v = w_u \otimes {w_g}'.$$

The weight ${w_g}'$ transforms the global variables and the parameters of g to reflect the behavior of the function. At the same time, ${w_g}'$ preserves the values of f's local variables and parameters.

Thus, the operation defined above *merges* the transformation of the local variables of the caller (which comes from the weight w_u) with the the transformation of the global variables and (output) parameters of the callee (which is constructed from both weights w_u and w_g). The parameters of g are later removed by the weights associated with the edges leading from g's return site, as shown in Eqn. (6.1).

6.4 Widening

In the previous section, we showed how to construct a domain of weights from a numeric abstract domain. However, the weight domains constructed from most existing numeric abstract domains fail to satisfy two of the conditions in the definition Defn. 6.3: namely, condition 2 ($\otimes$ must distribute over $\oplus$) and condition 4 (the weight domain must have no infinite descending chains). The non-distributivity condition is somewhat less important: as we mentioned earlier, existing WPDS techniques produce conservative (sound), although imprecise, results for non-distributive weight domains. Thus, we omit the detailed discussion of non-distributivity.[5]

The presence of infinite descending chains poses a much more significant problem: existing techniques for solving generalized pushdown reachability problems may not converge for such weight domains. To make these techniques work in practice, they have to be augmented with widening, much like the iterative techniques in Chapter 2. The WPDS library, which we used in our implementation, implements the techniques described in [97]: the widening approach we describe below is specific to these techniques, and may not work for other techniques for solving generalized pushdown reachability problems (e.g., [74]).

We construct two versions of the combine operation: a *regular* combine, denoted by $\oplus_r$, and a *widening* combine, denoted by $\oplus_w$. The regular combine is implemented as the join of the two input abstract-domain elements: $w_1 \oplus_r w_2 \triangleq w_1 \sqcup w_2$. The widening combine is slightly more complicated: it is defined as follows:

$$w_1 \oplus_w w_2 \triangleq w_1 \nabla (w_1 \sqcup w_2).$$

[5] The issue of non-distributivity is discussed to some degree in Chapter 7.

Note that, in contrast to the regular combine, the widening combine is *not* commutative. Thus, special care should be taken by the solver to feed the combine arguments in the proper order: generally, w_1 should be the “older” weight computed for a particular program point, whereas w_2 should be the “newer” weight, e.g., the weight computed for that program point after analyzing a loop iteration.

We identify the set of program points W, where the sequence of weights may form infinite descending chains. For that, we rely on the techniques proposed by Bourdoncle [18]: the set W contains the set of heads for all intraprocedural loops in the program. Also, for each interprocedural loop—i.e., for each loop in the call-graph of the program—we identify a single call-transition that breaks that loop and add *two* program points to the set W: the entry point of the callee, and the return site. To see why two program points need to be added, see Fig. 6.1: there are two loops—one formed by the nodes m_e, m_1, and m_2, the other formed by the nodes m_5, m_3, and m_4—that must be broken by widening points. For this example, the nodes m_e and m_3 are added to the set W.

Each iteration of the procedure for solving the GPS problem computes a new approximation for the weight at some program point by combining the weight approximation that has already been computed for that program point with the contribution made by one of the predecessors of that point: e.g., for a simple intraprocedural case, let $\langle u, v\rangle$ denote a control-flow edge, let w_u and w_v denote the weights at nodes u and v, and let $w_{\langle u,v\rangle}$ denote the weight associated with the edge $\langle u, v\rangle$; the new weight for the node v is computed as follows:

$$w_v{}' = w_v \oplus (w_u \otimes w_{\langle u,v\rangle}).$$

To incorporate widening, we modify the procedure as follows: if $v \in W$, we use the widening-combine operator $\oplus_w$ in place of $\oplus$; otherwise, we use the regular-combine operator $\oplus_r$. At the implementation level, rather than change the analysis engine, we extend the weight domain to incorporate certain annotations: the combine operation of the extended weight uses these annotations to choose whether to apply the regular combine or the widening combine to the underlying weights.

The need for widening also arises in the *path-summary* computation, which extracts the analysis results from the $\mathcal{W}$-automaton constructed by the procedure for solving GPS problem. The path-summary computation is similar in spirit to the intraprocedural analysis described in Chapter 2, where the control-flow graph of the program is given by the graph representation of the $\mathcal{W}$-automaton. Thus, the widening techniques from Chapter 2 can be used directly.

Chapter 7

Low-Level Library Analysis and Summarization

Static program analysis works best when it operates on an entire program. In practice, however, this is rarely possible. For the sake of maintainability and quicker development times, software is kept modular with large parts of the program hidden in libraries. Often, commercial off-the-shelf (COTS) modules are used. The source code for COTS components and libraries (such as the Windows dynamically linked libraries) is not usually available. Typically, to help program analysis deal with the absence of source code, library calls are modeled manually: either, by hard-coding the abstract transformers for selected library calls directly into the analyzer, or by providing function stubs that emulate certain aspects of the library calls in the language that the analyzer is able to understand. Manual modeling of library calls is time consuming, and it leaves open the question of whether the actual library implementation conforms to the models provided to the analyzer.

In this chapter, we take the first steps towards automating the process of modeling the effects of library functions. We present a technique that constructs automatically *summaries* for library functions by analyzing their low-level implementation (i.e., the library's binary). The "client" program analysis for which the summaries are constructed is memory-safety analysis [1, 37, 38, 107, 114]: the analysis that checks statically whether each memory accesses in the program is *safe*. We assume that the memory-safety analysis generates two types of error reports: buffer overruns and buffer underruns.

A library function's summary consists of a set of *error triggers* and a *summary transformer*. Error triggers are assertions over the program state that, if satisfied at the call site of the function, indicate a possibility of an error during the invocation of the function (e.g., a possibility of a buffer overrun or a buffer underrun, for the memory-safety analysis). A summary transformer specifies how the program state is affected by a function call; they are expressed as *transfer relations*, i.e., relations that hold among the values of global variables and function parameters *before* and *after* the call.

To use the function summaries, a client analysis must approximate the set of program states that reach the call site of a library function. The analysis should report a possible error if the approximation contains states that satisfy an assertion that corresponds to some error trigger. Summary transformers are applied as follows: the "before" values of global variables and function parameters are restricted to those that reach the call site; the restricted transfer relation is projected onto the "after" values to yield an approximation for the set of program states at the function's return point.

The application/library division provides a natural modularity border that should be useful to exploit for program-analysis purposes: typically, many applications link against the same library; summarizing the functions in that library obviates the need to reanalyze the library code for each application, which could improve analysis efficiency. (See §7.6 for a discussion of other work that has had the same motivation.)

Additionally, during development, application code is changed more frequently than library code. Because an application can be analyzed repeatedly against the same set of library summaries, it is possible to recoup the cost of applying more sophisticated (and thus, more expensive) analyses, such as polyhedral analysis [32] and shape analysis [78, 100], for library summarization.

Constructing summaries directly from the library implementation (as opposed to constructing them from the library specification) allows the client analysis to model precisely the deviations that that particular library implementation may have from its general specification (i.e., "bugs" and "features" in the library code). For instance, while experimenting with memory-management functions, we discovered that the standard C library implementation that came with Microsoft Developer Studio 6 assumes that the direction flag, the x86 flag that specifies the direction for string-manipulation instructions, is set to *false* on entry to the library. Thus, if a memory-management function (e.g., *memset*) is invoked in a program state in which the direction flag is set, the function does not behave in accordance with the specification (in fact, such an invocation causes the program to crash). If the summaries of memory-management functions constructed from this implementation of the library capture the relationship between the value of the direction flag and the behavior of the function, then the client analysis is able to alert the user about a potential problem when it detects that the direction flag is set to *true* at a call site of *memset*.

7.1 Overview of the Analysis

We use the function *memset* as the running example for this chapter. The function is declared as follows:

```
void * memset ( void * ptr, int value, size_t num );
```

Its invocation sets the first *num* bytes of the block of memory pointed to by *ptr* to the specified value (interpreted as an `unsigned char`). The value of *ptr* is returned.

As we suggested before, we address two types of memory-safety errors: buffer overruns and buffer underruns. Typically, analyses that target these types of errors propagate allocation bounds for each pointer. There are many ways in which this can be done. We use the following model. Two auxiliary integer variables are associated with each pointer variable: $alloc_f$ is the number of bytes that can be safely accessed beyond the address stored in the pointer variable (i.e., provides information about allowed positive offsets), $alloc_b$ is the number of bytes that can be safely accessed before the address stored in the pointer variable (i.e., provides information about allowed negative offsets). We believe that this scheme can be easily interfaced with other choices for representing allocation bounds. We use dot notation to refer to an allocation bound of a pointer variable, e.g., $ptr.alloc_f$.

7.1.1 Analysis Goals

A function summary specifies how to transform the program state at the call site of the function to the program state at its return site. Also, it specifies conditions that, if satisfied at the call site, indicate that a run-time error is possible during the function call. Intuitively, we expect the summary-transformer component of the function summary for *memset* to look like this (for the moment, we defer dealing with memory locations overwritten by *memset*—see §7.3.6):

$$ret = ptr \wedge ret.alloc_f = ptr.alloc_f \wedge ret.alloc_b = ptr.alloc_b, \tag{7.1}$$

where ret denotes the value that is returned by the function. We expect that a sufficient condition for the buffer overflow would look like this:

$$num \geq 1 \ \wedge \ ptr.alloc_f \leq num - 1. \tag{7.2}$$

That is, the value of num must be strictly greater than 0, otherwise memory is not accessed at all. Also, to cause the buffer overrun, the forward allocation bound $ptr.alloc_f$ must be strictly smaller than the number of bytes that are to be overwritten (i.e., the value of num). Similarly, a sufficient condition for the underflow should look like this:

$$num \geq 1 \ \wedge \ ptr.alloc_b \leq -1. \tag{7.3}$$

The goal of our analysis is to construct such function summaries automatically.

7.1.2 Analysis Architecture

Fig. 7.1 shows the disassembly of `memset` from the C library that is bundled with Visual C++.[1] Observe that there are no explicit variables in the code; instead, offsets from the stack register (`esp`) are used to access parameter values. Also, there is no type information, and thus it is not obvious which registers hold memory addresses and which do not. Logical instructions and shifts, which are hard to model numerically, are used extensively. Rather than addressing all these challenges at once, the analysis constructs the summary of a function in several phases.

Intermediate Representation (IR) Recovery. First, *value-set analysis (VSA)* [8, 9] is performed on the disassembled code to discover low-level information: variables that are accessed by each instruction, parameter-passing details, and, for each program point, an overapproximation of the values held in the registers, flags, and memory locations at that point. Also, VSA resolves the targets of indirect control transfers (indirect jumps and indirect calls).

[1] We used Microsoft Visual Studio 6.0, Professional Edition, *Release* build.

```
00401070   mov  edx, dword ptr [esp + 12]   edx ← count
00401074   mov  ecx, dword ptr [esp + 4]    ecx ← ptr
00401078   test edx, edx
0040107A   jz   004010C3                    if(edx = 0) goto 004010C3
0040107C   xor  eax, eax                    eax ← 0
0040107E   mov  al, byte ptr [esp + 8]      al ← (char)value
00401082   push edi
00401083   mov  edi, ecx                    edi ← ecx
00401085   cmp  edx, 4
00401088   jb   004010B7                    if(edx < 4) goto 004010B7
0040108A   neg  ecx                         ecx ← −ecx
0040108C   and  ecx, 3                      ecx ← ecx & 3
0040108F   jz   00401099                    if(ecx = 0) goto 00401099
00401091   sub  edx, ecx                    edx ← edx − ecx
00401093   mov  byte ptr [edi], al          ∗edi ← al
00401095   inc  edi                         edi ← edi + 1
00401096   dec  ecx                         ecx ← ecx − 1
00401097   jnz  00401093                    if(ecx ≠ 0) goto 00401093
00401099   mov  ecx, eax                    ecx ← eax
0040109B   shl  eax, 8                      eax ← eax << 8
0040109E   add  eax, ecx                    eax ← eax + ecx
004010A0   mov  ecx, eax                    ecx ← eax
004010A2   shl  eax, 10h                    eax ← eax << 16
004010A5   add  eax, ecx                    eax ← eax + ecx
004010A7   mov  ecx, edx                    ecx ← edx
004010A9   and  edx, 3                      edx ← edx & 3
004010AC   shr  ecx, 2                      ecx ← ecx >> 2
004010AF   jz   004010B7                    if(ecx = 0) goto 004010B7
004010B1   rep stosd                        while (ecx ≠ 0) {
                                               ∗edi ← eax; edi++; ecx--; }
004010B3   test edx, edx
004010B5   jz   004010BD                    if(edx = 0) goto 004010BD
004010B7   mov  byte ptr [edi], al          ∗edi ← al
004010B9   inc  edi                         edi ← edi + 1
004010BA   dec  edx                         edx ← edx − 1
004010BB   jnz  004010B7                    if(edx ≠ 0) goto 004010B7
004010BD   mov  eax, dword ptr [esp + 8]    eax ← ptr
004010C1   pop  edi
004010C2   retn                             return
004010C3   mov  eax, dword ptr [esp + 4]    eax ← ptr
004010C7   retn                             return
```

Figure 7.1 The disassembly of *memset*. The rightmost column shows the semantics of each instruction using a C-like notation.

In x86 executables, parameters are typically passed via the stack. The register `esp` points to the top of the stack and is implicitly updated by `push` and `pop` instructions. VSA identifies numeric properties of the values stored in `esp`, and maps offsets from `esp` to the corresponding parameters. To see that this process is not trivial, observe that different offsets map to the same parameter at addresses *0x4010BD* and *0x4010C3*: at *0x4010BD* an extra 4 bytes are used to account for the push of `edi` at *0x401082*.

Numeric-Program Generation. VSA results are used to generate a numeric program that captures the behavior of the library function. The primary challenge that is addressed in this phase is to translate non-numeric instructions, such as bitwise operations and shifts, into a program that a numeric analyzer is able to analyze. Bitwise operations are used extensively in practice to perform certain computations because they are typically more efficient in terms of CPU cycles than corresponding numeric instructions. A ubiquitous example is the use of the `xor` instruction to initialize a register to zero. In Fig. 7.1, the `xor` at *0x40107C* is used in this way. The `test` instruction updates the x86 flags to reflect the application of *bitwise and* to its operands (the operands themselves are not changed). Compilers often use `test` to check whether the value of a register is equal to 0 (see the instructions at addresses *0x401078* and *0x4010B3* in Fig. 7.1).

A more complicated case is when several instructions, neither of which can be modeled precisely with the numeric abstraction we use, cooperate to establish a numeric property that is relevant to the analysis. In Fig. 7.1, the two consecutive instructions at *0x4010A9*, a *bitwise and* and a *right shift*, cooperate to establish the property

$$edx_0 = 4 \times ecx + edx,$$

where edx_0 denotes the value stored in register `edx` before the instructions are executed. Note that the property itself can be expressed with the numeric abstraction we use (the polyhedral abstract domain), but it is impossible to capture it by considering each instruction in isolation. We describe how we handle this situation in §7.3.5.

The numeric-program-generation phase also introduces the auxiliary variables that store allocation bounds for pointer variables. A simple type inference is performed to identify variables and registers that may hold memory addresses. For each instruction that performs address arithmetic, additional statements that update corresponding allocation bounds are generated. Also, for each instruction that dereferences an address, a set of numeric assertions are generated to ensure that memory safety is not violated by the operation. The assertions divert program control to a set of specially-introduced *error program points*: two sink[2] nodes are introduced into the program's CFG for each memory access that is checked—one node for the buffer overflow, the other for the buffer underflow.

Fig. 7.2 shows the numeric program that is generated for *memset*.

Numeric Analysis and Summary Construction. The generated numeric program is fed into our WPDS-based numeric program analyzer. The analyzer computes, for each program point, a function that maps an approximation for the set of initial states at the entry of the program to an approximation for the set of states that arise at that program point. The numeric-analysis results are used to generate a set of error triggers and a set of summary transformers for the library function. The transfer functions computed for program points that correspond to return instructions form a set of summary transformers for the function. Error triggers are constructed by projecting transfer functions computed for the set of error program points onto their domains.

One challenge posed by the numeric analysis is that the polyhedral abstract domain, which is employed by our analyzer, does not scale well as the number of program variables that need to be modeled increases. We address this issue by using an existing technique for improving scalability of numeric analysis: *variable packing* [87]. The idea behind this technique is to identify groups of related variables (referred to as *packs*) and to track numeric relationships in each group individually. That is, instead of a single polyhedron with a large number of dimensions, a larger number of lesser-dimensional polyhedra are propagated by the analysis. The groups of variables need not be disjoint, and some program variables may not be in any group at all. We identify the groups of related variables by tracking variable dependences, such as data and control dependences. The detailed description of our use of variable packing is presented in §7.4.1.

[2] A *sink* node is a node that has no outgoing edges.

```
                    memset($(ptr, value, num)$)
00401070                 $edx \leftarrow count;$
00401074                 $ecx \leftarrow ptr; ecx.alloc_f \leftarrow ptr.alloc_f; ecx.alloc_b \leftarrow ptr.alloc_b;$
00401078-7A              if$(edx = 0)$ goto $L5;$
0040107C-82              ...
00401083                 $edi \leftarrow ecx; edi.alloc_f \leftarrow ecx.alloc_f; edi.alloc_b \leftarrow ecx.alloc_b;$
00401088                 if$(edx < 4)$ goto $L3;$
0040108A                 $ecx \leftarrow -ecx;$
0040108C                 $ecx \leftarrow ?;$ assume$(0 \leq ecx \leq 3);$
0040108F                 if$(ecx = 0)$ goto $L2;$
00401091                 $edx \leftarrow edx - ecx;$
00401093            L1:  assert$(edi.alloc_f >= 1);$ assert$(edi.alloc_b >= 0);$
00401095                 $edi \leftarrow edi + 1;\ edi.alloc_f \leftarrow edi.alloc_f - 1;\ edi.alloc_b \leftarrow edi.alloc_b + 1;$
00401096                 $ecx \leftarrow ecx - 1;$
00401097                 if$(ecx \neq 0)$ goto $L1;$
00401099-A5         L2:  ...
004010A7                 $edx.rem_4 = ?;\ edx.quot_4 = ?;$
                         assume$(0 \leq edx.rem_4 \leq 3);$ assume$(edx = 4 \times edx.quot_4 + edx.rem_4);$
                         $ecx \leftarrow edx;\ ecx.quot_4 \leftarrow edx.quot_4;\ ecx.rem_4 = edx.rem_4;$
004010A9                 $edx \leftarrow edx.rem_4;$
004010AC                 $ecx \leftarrow ecx.quot_4;$
004010AF                 if$(ecx = 0)$ goto $L3;$
004010B1                 assert$(edi.alloc_f >= 4 \times ecx);$ assert$(edi.alloc_b >= 0);$
                         $edi \leftarrow edi + 4 \times ecx;$
                         $edi.alloc_f \leftarrow edi.alloc_f - 4 \times ecx;\ edi.alloc_b \leftarrow edi.alloc_b + 4 \times ecx;$
                         $ecx \leftarrow 0;$
004010B3-B5              if$(edx = 0)$ goto $L4;$
004010B7            L3:  assert$(edi.alloc_f >= 1);$ assert$(edi.alloc_b >= 0);$
004010B9                 $edi \leftarrow edi + 1;\ edi.alloc_f \leftarrow edi.alloc_f - 1;\ edi.alloc_b \leftarrow edi.alloc_b + 1;$
004010BA                 $edx \leftarrow edx - 1$
004010BB                 if$(edx \neq 0)$ goto $L3;$
004010BD            L4:  $eax \leftarrow ptr;\ eax.alloc_f = ptr.alloc_f;\ eax.alloc_b \leftarrow ptr.alloc_b;$
004010C2                 return $eax, eax.alloc_f, eax.alloc_b;$
004010C3            L5:  $eax \leftarrow ptr;\ eax.alloc_f = ptr.alloc_f;\ eax.alloc_b \leftarrow ptr.alloc_b;$
004010C7                 return $eax, eax.alloc_f, eax.alloc_b;$
```

Figure 7.2 The numeric program generated for the code in Fig. 7.1; parts of the program that are not relevant for the summary construction are omitted from the listing shown above.

7.1.3 The summary obtained for *memset*

The implementation of *memset* uses two loops and a "`rep stosd`" instruction, which invokes a hardware-supported loop. The "`rep stosd`" instruction at *0x4010B1* is the workhorse; it performs the bulk of the work by copying the value in `eax` (which is initialized in lines *0x40107C–0x40107E* and *0x401099–0x4010A5* to contain four copies of the low byte of *memset*'s $value$ parameter) into successive 4-byte-aligned memory locations. The loops at *0x401093–0x401097* and

0x4010B7–0x4010BB handle any non-4-byte-aligned prefix and suffix. If the total number of bytes to be initialized is less than 4, control is transfered directly to the loop at *0x4010B7*.

The application of our technique to the code in Fig. 7.1 yields exactly the summary transformer we conjectured in Eqn. (7.1). The situation with error triggers is slightly more complicated. First, observe that there are three places in the code where the buffer is accessed: at addresses *0x401093*, *0x4010B1*, and *0x4010B7*. Each access produces a separate error trigger:

	Buffer overrun	Buffer underrun
0x401093	$num \geq 4 \ \wedge \ ptr.alloc_f \leq 2$	$num \geq 4 \ \wedge \ ptr.alloc_b \leq -1$
0x4010B1	$num \geq 4 \ \wedge \ ptr.alloc_f \leq num - 1$	$num \geq 4 \ \wedge \ ptr.alloc_b \leq -1$
0x4010B7	$num \geq 1 \ \wedge \ ptr.alloc_f \leq num - 1$	$num \geq 1 \ \wedge \ ptr.alloc_b \leq -1$ $\wedge \, ptr.alloc_b \leq 2 - num$

Note that the first buffer-overrun trigger is stronger than the one conjectured in Eqn. (7.2): it gives a constant bound on $ptr.alloc_f$; furthermore, the bound is less than 3, which is the smallest bound implied by the conjectured trigger for $num \geq 4$ (see Eqn. (7.2)). The issue is that the instruction at *0x401093* is executed only if the number of bytes to be overwritten (num) is greater than 4, and accesses at most three first bytes of the buffer pointed to by ptr (the actual number of bytes that are accessed depends on the alignment of the memory address in ptr). Thus, a buffer overrun at *0x401093* can only happen if the forward allocation bound for ptr is less than 3. In cases where $num \geq 4$ and $ptr.alloc_f$ is equal to 3, *memset* will generate a buffer overrun not at *0x401093*, but at one of the other two memory accesses instead.

The other two buffer-overrun triggers are similar to the trigger conjectured in Eqn. (7.2) and differ only in the value of num. The buffer-underrun triggers are similar to the trigger that was conjectured in Eqn. (7.3), except for the trigger generated for the memory access at *0x4010B7*. That trigger contains an extra constraint $ptr.alloc_b \leq 2 - num$, which indicates that, by the time the control gets to *0x4010B7*, the memory address that was originally specified in ptr has been advanced by at least $num - 3$ bytes.

Note that, although the triggers shown above provide error conditions that are sufficiently precise for the client analysis to avoid generating spurious error reports (false positives), these triggers are somewhat imprecise. For instance, if the value of variable num is 5 and the forward allocation bound $ptr.alloc_f$ is 1, then the above triggers indicate that the buffer overrun can happen at any of the three memory accesses. For the purpose of providing better diagnostic information, however, it may be advantageous to link the buffer overrun to the particular memory access that could generate it. In the situation above, the memory access at which the buffer overrun occurs is determined by the alignment of the pointer ptr: if ptr is either 4-byte aligned or 3 bytes off (i.e., $ptr \equiv 3$ **mod** 4), the buffer overrun occurs at *0x4010B1*; if ptr is either 1 or 2 bytes off, the buffer overrun occurs at *0x401093*; the buffer overrun never occurs at *0x4010B7*. This imprecision in the triggers is due to the conservative translation of the instructions that check pointer alignment into the corresponding numeric statements. In §7.3.5, we present a technique that allows pointer alignment to be modeled more precisely, resulting in better error triggers.

The next several sections present the phases of the analysis outlined above in greater detail. Particular attention is paid to the numeric-program-generation phase, which is the primary contribution of our work.

7.2 Intermediate-Representation Recovery

The IR-recovery phase recovers intermediate representations from the library's binary that are similar to those that would be available had one started from source code. For this phase, we use the CodeSurfer/x86 analyzer that was developed jointly by Wisconsin and GrammaTech, Inc. This tool recovers IRs that represent the following information:

- control-flow graphs (CFGs), with indirect jumps resolved;
- a call graph, with indirect calls resolved;
- information about the program's variables;
- possible values of pointer variables;
- sets of used, killed, and possibly-killed variables for each CFG node; and

- data dependences.

The techniques employed by CodeSurfer/x86 do not rely on debugging information being present, but can use available debugging information (e.g., Windows .pdb files) if directed to do so.

The analyses used in CodeSurfer/x86 (see [8, 9]) are a great deal more ambitious than even relatively sophisticated disassemblers, such as IDAPro [61]. At the technical level, they address the following problem: *Given a (possibly stripped) executable E (i.e., with all debugging information removed), identify the procedures, data objects, types, and libraries that it uses, and,*

- *for each instruction I in E and its libraries,*
- *for each interprocedural calling context of I, and*
- *for each machine register and variable V,*

statically compute an accurate over-approximation to the set of values that V may contain when I executes.

7.2.1 Variable and Type Discovery.

One of the major stumbling blocks in analyzing executables is the difficulty of recovering information about variables and types, especially for aggregates (i.e., structures and arrays).

When debugging information is absent, an executable's data objects are not easily identifiable. Consider, for instance, a data dependence from statement `a` to statement `b` that is transmitted by write/read accesses on some variable `x`. When performing source-code analysis, the programmer-defined variables provide us with convenient compartments for tracking such data manipulations. A dependence analyzer must show that `a` defines `x`, `b` uses `x`, and there is an `x`-def-free path from `a` to `b`. However, in executables, memory is accessed either directly—by specifying an absolute address—or indirectly—through an address expression of the form "[*base* + *index* × *scale* + *offset*]", where *base* and *index* are registers and *scale* and *offset* are integer constants. It is not clear from such expressions what the natural compartments are that should be used for analysis. Because, executables do not have *intrinsic* entities that can be used for analysis (analogous to source-level variables), a crucial step in IR recovery is to identify variable-like entities.

The variable and type-discovery phase of CodeSurfer/x86 [9], recovers information about variables that are allocated globally, locally (i.e., on the run-time stack), and dynamically (i.e., from the heap). An iterative strategy is used; with each round of the analysis—consisting of aggregate structure identification (ASI) [9, 93] and value-set analysis (VSA) [8, 9]—the notion of the program's variables and types is refined. The net result is that CodeSurfer/x86 recovers a set of proxies for variables, called *a-locs* (for "abstract locations"). The a-locs are the basic variables used in the method described below.

7.3 Numeric-Program Generation

The generation of a numeric program is the central contribution of our technique. The target language for the numeric program corresponds to the language we described in Chapter 2 with the exception that programs with multiple procedures can be generated. The language supports assignments, assumes, asserts, if-statements, procedure calls, and gotos. The expression "?" selects a value non-deterministically. The condition "*" transfers control non-deterministically.

The generation process abstracts away some aspects of the binary code that cannot be modeled precisely in the polyhedral abstract domain; thus, the generated program is usually non-deterministic and cannot be directly executed. Note that this represents a slight deviation from the discussion in Chapter 2, where we relied on the abstract domain to conservatively handle arbitrary numeric and conditional expressions. For this application, we chose to deal with the expressions that cannot be modeled precisely by the polyhedral domain at the level of numeric-program generation. This gave us more flexibility in designing techniques that improve the overall precision of the analysis.

We strive as much as possible to generate a sound representation of the binary code. However, the current implementation of the analysis assumes that numeric values are unbounded. In the future, we hope to add support for bounded arithmetic.

7.3.1 Numeric-Program Variables

The *initial* set of numeric-program variables is constructed from the results obtained by value-set analysis (VSA): a numeric variable is created for each 4-byte a-loc identified by VSA. We only consider 4-byte a-locs because only those a-locs can store 32-bit memory addresses, which we need to model for the memory-safety analysis. The eight x86 registers: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, and `esp`, are modeled by global variables. A number of additional variables—to keep track of allocation bounds, to model memory locations that are not resolved by the VSA, and to handle integer division and remainder computations—are introduced as described in the rest of this section.

An operand of x86 instruction can be either an immediate value, a register, or a memory location specified via one of the x86 memory-addressing modes. We map the x86 operands to the numeric-program operands as follows:

- **Immediate values.** The generated numeric statement uses the immediate value that is specified by the instruction.

- **Registers.** If the register is one of the eight registers that we model, the generated numeric statement uses the global variable that corresponds to that register; otherwise, if it is a register that we do not model, there are two cases: (i) if it is the *target* operand (that is, the operand that is updated by the instruction), the instruction is translated into the numeric program's equivalent of a no-op, (ii) if it is a source operand, the non-deterministic value "?" is used in the generated numeric statement.

- **Memory operands.** The VSA results are queried to get a set of a-locs that the memory operand may refer to. There are two possible answers: either (i) the memory operand is *resolved*, in which case a set of a-locs is returned, or (ii) the memory operand is *unresolved*, in which case the value $\top$ is returned.

  - *Resolved operands.* The set of a-locs constructed by VSA may contain *full* a-locs (that is, the memory operand refers to the entire a-loc), and *partial* a-locs (that is, the memory operand refers only to a part of the a-loc). If the operand is updated by the instruction, numeric-update statements are generated for the numeric variables that corresponds to each full 4-byte a-loc in the set, and *forget* statements[3] are generated for the numeric variables that correspond to each 4-byte partial a-loc.

If the operand is used, but not updated, then there are two possibilities: either (i) the operand is determined exactly (i.e., the set contains a single full 4-byte a-loc), in which case the numeric variable that corresponds to that a-loc is used in the generated numeric statement; or (ii) the set contains more than one a-loc, in which case a non-deterministic value $?$ is used in the generated numeric statement.

– *Unresolved operands.* The unresolved memory operands are modeled by *symbolic memory constants*; that is, a global variable is created for each unresolved memory operand to model symbolically the value at the accessed memory location. That global variable is used in the generated numeric statement. We describe symbolic memory constants in more detail in §7.3.6

We will use a function $\mathit{Var}_{\mathit{NP}}$ to map the operands of x86 instructions to the corresponding numeric variables.

7.3.2 Basic Translation of x86 Instructions

In the previous section, we explained how instruction operands are mapped to the corresponding numeric variables. In this section, we briefly outline the basic translation of x86 instructions into the corresponding numeric statements. The goal of the basic translation is to capture the “natural” semantics of each instruction. We address the effect of the instructions on auxiliary variables (e.g., allocation bounds) in §7.3.4 and §7.3.5.

For most instructions, the translation is straightforward. Simple instructions, such as `mov`, `add`, `sub`, `lea`, etc., are directly translated into the corresponding numeric statements: e.g., the

[3] Recall from Chapter 2 that the *forget* statement $x \leftarrow ?$ assigns a non-deterministically chosen value to the target variable x.

instruction “`sub edx,ecx`” at *0x401091* in Fig. 7.1 is translated into numeric statement $edx \leftarrow edx - ecx$.

Bitwise operations and shifts typically cannot be converted precisely into a single numeric statement, and thus pose a greater challenge. Several numeric statements, including ifs and assumes, may be required to translate each of these instructions. At first, we were tempted to design universal translations that would work equally well for all possible contexts in which the instruction occurs. In the end, however, we noticed that these instructions, when employed in numeric computations, are only used in a few very specific ways. For instance, bitwise-and is often used to compute the remainder from dividing a variable by a power of two. The instruction “`and ecx,3`” at *0x40108C* in Fig. 7.1 is used to compute ecx **mod** 4. The translation treats these special cases with precision; other cases are treated imprecisely, but soundly.

Below we show how the `and` and `or` instructions are translated into numeric program statements. The translation of an instruction of the form “`and` $op_1,\ op_2$” recognizes the special case of op_2 being an immediate value that is greater than zero:

$$\texttt{and}\ op_1,\ op_2 \Rightarrow \begin{cases} \left[\begin{array}{l} \mathit{Var_{NP}}(op_1) \leftarrow ?; \\ \textbf{assume}(0 \le \mathit{Var_{NP}}(op_1) \le \mathit{Var_{NP}}(op_2)); \end{array}\right] & \text{if } op_2 > 0 \text{ is an imm. value} \\ \mathit{Var_{NP}}(op_1) \leftarrow ?; & \text{otherwise} \end{cases}$$

The translation of an instruction of the form “`or` $op_1,\ op_2$” recognizes two special cases: (i) the case of op_2 being an immediate value *0x0xFFFFFFFF* (-1 in 2’s complement arithmetic), and (ii) the case in which both operands are the same:

$$\texttt{or}\ op_1,\ op_2 \Rightarrow \begin{cases} \mathit{Var_{NP}}(op_1) \leftarrow (-1); & \text{if } op_2 = \mathit{0xFFFFFFFF} \\ \textbf{nop}; & \text{if } op_1 = op_2 \\ \mathit{Var_{NP}}(op_1) \leftarrow ?; & \text{otherwise} \end{cases}$$

Hardware-supported loops. An interesting class of instructions is the x86 hardware-supported loops, such as the instruction “`rep stosd`” at address *0x4010B1* in Fig. 7.1. The “`stosd`” instruction writes the value stored in register `eax` into a memory location specified by the address stored in register `edi`, and advances the address in `edi` either forwards (i.e., $edi \leftarrow edi + 4$) if the direction flag *DF* is cleared (i.e., $\mathit{DF} = \mathit{false}$), or backwards (i.e., $edi \leftarrow edi - 4$) if the direction flag is set (i.e., $DF = true$). The prefix "`rep`" repeats the "`stosd`" instruction until the values stored in register `ecx` becomes zero: `ecx` is decremented on each iteration.[4].

The current implementation translates the instruction "`rep stosd`" as follows: the intermediate representation is queried for the value of the direction flag: if the flag is *false,* the instruction is translated into the following set of numeric statements: "$edi \leftarrow edi + 4 \times ecx;\ ecx \leftarrow 0;$"; if the flag is *true,* the instruction is translated into "$edi \leftarrow edi - 4 \times ecx;\ ecx \leftarrow 0;$"; if the intermediate representation cannot determine the state of the flag definitely (i.e., if the flag can be either *true* or *false*), the instruction is translated as follows:

$$\begin{aligned}&\textbf{if}(*)\ edi \leftarrow edi + 4 \times ecx;\ \textbf{else}\ edi \leftarrow edi - 4 \times ecx;\\ &ecx \leftarrow 0;\end{aligned} \tag{7.4}$$

Other instructions that execute hardware-supported loops are translated similarly. The only complication is caused by the instructions "`rep cmpsd`" and "`rep scasd`": these instructions may exit the loop before the value stored in register `ecx` reaches zero: on each iteration, the value of the x86 zero flag *ZF* is checked—if the instruction prefix is "`repe`", the instruction checks whether the flag is *true*; if the prefix is "`repne`", the instruction checks whether the flag is *false*—and if the flag is set accordingly, the loop is terminated. To translate these instructions, we introduce an extra numeric variable for which the value is selected non-deterministically from the range $[1, ecx]$: this variable models the number of iterations performed by the loop. For instance, if the direction flag is *false,* the instruction "`rep cmpsd`" is translated into the following sequence of numeric-program statements:

$$\begin{aligned}&temp =?;\ \textbf{assume}(1 \le temp \le ecx);\\ &edi \leftarrow edi + 4 \times temp;\ esi \leftarrow esi + 4 \times temp;\\ &ecx \leftarrow ecx - temp;\end{aligned}$$

Let us briefly discuss the direction-flag issue mentioned in the introduction to this chapter. The particular implementations of memory functions that we experimented with, such as `memset`, assume that the direction flag is *false* on the entry to the function; i.e., these implementations do not clear the flag explicitly. As a result, the IR-recovery phase presumes that the value of the

[4] When "`stosd`" appears without prefix "`rep`", register `ecx` is not decremented.

direction flag is unknown at the program point where a particular x86 hardware-loop instruction is executed. Consequently, that instruction is translated according to Eqn. (7.4). Note that this translation is overly conservative in the sense that the function summary synthesized from this translation is independent from the value of the direction flag at the call-site of the function; i.e., the client analysis may generate spurious error reports in case the direction flag is indeed *false* at the call site. In our experiments, we dealt with this issue by making the same assumption that the implementation of the functions makes: that is, that the direction flag is *false* on entry to the function. However, this assumption is unsound. A better way of handling this issue is to produce two function summaries: one for the case where the direction flag is *true* at the call site, the other for the case where it is *false* at the call site. Then, the client analysis may use its own approximation for the value of the direction flag to select the function summary to be applied.

Recovering conditions from the branch instructions. An important part of numeric-program generation is the recovery of conditional expressions. In the x86 architecture, several instructions must be executed in sequence to perform a conditional control transfer. The execution of most x86 instructions affects the set of flags maintained by the processor. The flags include the *zero flag*, which is set if the result of the currently executing instruction is zero, the *sign flag*, which is set if the result is negative, and many others. Also, the x86 architecture provides a number of control-transfer instructions, each of which performs a jump if the flags are set in a specific way. Technically, the flag-setting instruction and the corresponding jump instructions do not have to be adjacent and can, in fact, be separated by a set of instructions that do not affect the flags (such as the `mov` instruction.

We symbolically propagate the expressions that affect flags to the jump instructions that use them. Consider the following sequences of instructions and their translation:

`cmp eax,ebx`	
`mov ecx,edx`	$ecx \leftarrow edx;$
`jz label`	$\textbf{if}(eax - ebx = 0)\ \textbf{goto}\ label;$

We derive a flag-setting expression $eax - ebx$ from the `cmp` instruction; the `mov` instruction does not affect any flags; the `jz` instruction transfers control to `label` if the zero flag is set, which can only happen if the expression $eax - ebx$ is equal to zero. Note, however, that if the intervening move affects one of the operands in the flag-setting expression, that expression is no longer available at the jump instruction. This can be circumvented with the use of a temporary variable:

`cmp eax,ebx`	
`mov eax,edx`	$temp \leftarrow eax - ebx;\ eax \leftarrow edx;$
`jz label`	**if**$(temp = 0)$ **goto** $label;$

7.3.3 Value Dependence Graph

To aid in the generation of numeric programs, we construct an auxiliary data structure, which we call the *value dependence graph (VDG)*. Intuitively, the graph captures the dependences among values stored in registers and a-locs identified by the IR-recovery phase (we will refer to these collectively as *x86 variables*). The dependence information is used for the following purposes:

- To identify variables that are used to store and propagate memory addresses. These variables must have auxiliary variables that represent allocation bounds associated with them (see §7.3.4).

- To identify variables that are used to store and propagate values to which an integer division and a remainder computation are applied. These variables are also associated with a certain set of auxiliary variables, which allow the precision of polyhedral analysis to be improved (see §7.3.5).

- To identify variable packs (that is groups of related variables) for polyhedral analysis. Recall from §7.1.2 that the polyhedral analysis scales poorly as the number of variables that have to be tracked increases. Variable packing is a technique for improving the scalability of the analysis (see §7.4.1).

The nodes of the graph correspond to the x86 variables, and edges represent dependences among them. The graph is constructed by taking a single path through the program and introducing corresponding nodes and edges for each instruction. More formally:

VDG nodes. Ideally, we would like to have a separate node for each *role* of an x86 variable (e.g., for each class of values that the variable stores). While a-locs that correspond to the "natural" global and local variables of the analyzed program are likely to play the same role throughout their lifetime, registers and a-locs that correspond to stack locations accesses by the `push` and `pop` instructions[5] generally play many different roles. That is, the same register may be used to store both numeric values and memory addresses, and the same location on the stack can be used to pass parameters of different types to different functions.

We create a single graph node for each a-loc that corresponds to a local or global variable and for each symbolic memory constant. To distinguish among the multiple roles played by registers and stack locations, we create a graph node for each definition and a graph node for each use of a register or a stack location. That is, a separate node is created for each instruction that uses or defines a register, and two nodes are created for the instructions that both use and define a register. For example, instruction "`add eax, ebx`" generates three graph nodes: two for the uses of registers `eax` and `ebx`, and one for the definition of `eax`. For stack locations, the definition nodes are created by `push` instructions, and the use nodes are created by function calls and `pop` instructions.[6]

For each instruction, there is a unique mapping of its operands to the corresponding graph nodes. We define two functions to represent this mapping:

$$\mathit{Node}_D : \mathit{Instr} \times \mathit{Operand} \rightarrow \mathit{Node} \qquad \text{and} \qquad \mathit{Node}_U : \mathit{Instr} \times \mathit{Operand} \rightarrow \mathit{Node}$$

That is, for the instruction I, $\mathit{Node}_D(I, op_1)$ gives the node corresponding to the new definition of the first operand create by I, and $\mathit{Node}_U(I, op_2)$ gives the node corresponding to the use of the second operand. We also define a function that maps the nodes of the VDG back to the corresponding a-locs, registers, and symbolic memory constants: Var_{x86}.

[5] These stack locations are used for two main purposes: (i) to store register values during function calls, and (ii) to pass function parameters.

[6] Some programs manipulate the stack directly, that is without using `push` and `pop` instructions. Our current implementation does not handle such programs yet, but can be trivially extended to do so.

VDG edges. Edges are added to reflect the dependences between values induced by x86 instructions. We distinguish among several types of edges (the edges are explicitly labeled with their corresponding types):

- *Affine-dependence edges:* an affine-dependence edge is added to the graph if the value of the x86 variable at the destination of the edge is computed as an affine transformation of the value of the x86 variable at the source of the edge. That is, if the effect of the instruction I on its operands is:

  $$op_i \leftarrow op_j + c_1 \times op_k + c_2,$$

  where c_1 and c_2 are constants, then an affine-dependence edge from $\textit{Node}_U(I, op_j)$ to $\textit{Node}_D(I, op_i)$ is added to the graph. Affine-dependence edges are induced by `mov`, `add`, `sub`, `inc`, `dec`, `lea`, `push`, `pop`, etc. Also, affine-dependence edges are used to connect the nodes representing stack locations that store the values of actual parameters at a function call site to the nodes that represent the formal parameters of the function, and to connect the nodes that represent the register `eax` at a function return statement to the node that represents the register `eax` at the call site's return point.

- *Non-affine-dependence edges:* a non-affine-dependence edge is added to the graph if the value of the variable at the source of the edge contributes to the computation of the value of the variable at the destination of the edge, but the dependence cannot be qualified as affine. For the instruction I in the previous bullet point, a non-affine-dependence edge from $\textit{Node}_U(I, op_k)$ to $\textit{Node}_D(I, op_i)$ is added to the graph. Non-affine-dependence edges are induced by most of the same instructions as the affine flow edges, plus the instructions `and`, `shl`, `shr`, `sar`, etc.

- *Conditional-dependence edges:* conditional-dependence edges represent the dependences that are induced by the instructions that evaluate conditions. These edges are generated by the `cmp` and `test` instructions. For instance, the instruction I, "`cmp` op_1, op_2", generates two edges, in opposite directions, between the nodes $\textit{Node}_U(I, op_1)$ and $\textit{Node}_U(I, op_2)$.

- *Loop-dependence edges:* loop-dependence edges capture dependences between loop induction variables and variables that appear in loop-exit conditions. These edges are not generated by a particular x86 instruction, rather several instructions cooperate to create a single loop-dependence edge. To generate these edges, a set of variables that are incremented in the body of the loop and a set of variables that appear in loop-exit conditions are collected for each loop. A loop-dependence edge is added to the graph from each node that represents an incremented variable to each node that represents a loop-exit-condition variable.

The affine-dependence edges are used for identifying variables that (i) store memory addresses, or (ii) variables that participate in integer computations, such as integer division and remainder computations. All dependence edges are used in the algorithm for variable-pack identification.

Implicit dependencies. Due to the choice to introduce a unique VDG node for each definition and each use of registers and stack locations, certain dependences cannot be recovered by interpreting individual instructions. In particular, the edges that link definitions of registers and stack locations to their subsequent uses need to be added to the graph. The following sequence of instructions provides an example:

```
mov eax, 5
mov ebx, eax
```

Neither of the two instructions generates an edge from the node that corresponds to the definition of `eax` created by the first instruction to the node that corresponds to the use of `eax` created by the second instruction. The missing dependence edges can be generated by performing a simple intraprocedural reaching-definitions analysis for registers and stack locations, and adding affine-dependence edges from each definition node to each use node reached by that definition.

7.3.4 Memory-Safety Checks and Allocation Bounds

As we mentioned before, each numeric-program variable var that may contain a memory address is associated with two auxiliary variables that specify allocation bounds for that address. The auxiliary variable $var.alloc_f$ specifies the number of bytes following the address that can be safely accessed; the auxiliary variable $var.alloc_b$ specifies the number of bytes preceding the address that can be safely accessed. These auxiliary variables are central to our technique: the purpose of numeric analysis is to infer constraints on the auxiliary variables that are associated with the function's parameters, global variables, and return value. These constraints form the bulk of the function summaries.

Memory-Safety Checks. The auxiliary variables are used to generate memory-safety checks: checks for buffer overflows and checks for buffer underflows. We generate memory-safety checks for each memory access that is not resolved by the IR-recovery phase to a particular global or local variable: these memory accesses correspond to accesses to global and local buffers, and to dereferences of pointers that are either passed as parameters or stored in global variables. As mentioned in §7.2, general indirect memory accesses in x86 instructions have the form "[*base* + *index* × *scale* + *offset*]", where *base* and *index* are registers and *scale* and *offset* are constants. Let *size* denote the number of bytes to be read or written. The following checks are generated:

- Buffer-overflow check: **assert**$(base.alloc_f \geq index \times scale + \textit{offset} + \textit{size})$
- Buffer-underflow check: **assert**$(base.alloc_b + index \times scale + \textit{offset} \geq 0)$

The instructions that execute hardware-supported loops require slightly more care: the value of the x86 direction flag and the number of iterations must be taken into consideration. For instance, for the the instruction "`rep stosd`", which we described in §7.3.2, the following checks are generated:

	Buffer overrun	Buffer underrun
$DF = \textit{false}$	**assert**$(edi.alloc_f \geq 4 \times ecx)$	**assert**$(edi.alloc_b \geq 0)$
$DF = \textit{true}$	**assert**$(edi.alloc_f \geq 0)$	**assert**$(edi.alloc_b \geq 4 \times ecx)$
$DF = \textit{unknown}$	**assert**$(edi.alloc_f \geq 4 \times ecx)$	**assert**$(edi.alloc_b \geq 4 \times ecx)$

Some instructions, namely "`rep movsd`" and "`rep cmpsd`", require that memory-safety checks are generated for both of its implicit operands: register `edi` and register `esi`. Also, note that the discussion regarding the modeling of the direction flag at the end of §7.3.2 applies to the case of generating memory-safety checks, too.

Allocation bounds. Maintaining allocation bounds for all variables is unnecessarily expensive. For this reason, we only associate allocation bounds with variables that can hold memory addresses. To identify this set of variables, we use the value-dependence graph, which was described in §7.3.3. We perform a backward traversal of the graph, starting from the set of nodes that are guaranteed to represent pointer variables (i.e., they are dereferenced by some x86 instructions). The nodes that are visited by the traversal are collected into the resulting set, which we denote by *Addr*.

More formally, the initial approximation for the set *Addr* contains the VDG nodes that represent variables that are treated as pointers in the memory-safety checks we introduced. For instance, for the instruction

$$I\colon \texttt{mov eax, [esi + 4} \times \texttt{ecx + 4]}$$

we add $\mathit{Node}_U(I, esi)$ to the initial approximation for *Addr*. The set *Addr* is iteratively grown by adding the nodes that reach the nodes in the set via affine-dependence edges. The process stops when no more nodes can be added.

The updates for the auxiliary variables are generated in a straightforward way. That is, the translation of the `mov` instruction contains assignments for the corresponding allocation bounds. The translations of `add`, `sub`, `inc`, `dec`, and `lea`, as well as the x86 string-manipulation instructions, contain affine transformations of the corresponding allocation bounds (see Figs. 7.1 and 7.2 for some examples).

7.3.5 Integer Division and Remainder Computations

Memory functions, such as `memset`, rely heavily on integer division and remainder computations to improve the efficiency of memory operations. In low-level code, the quotient and remainder from dividing by a power of two are typically computed with a shift-right (`shr`) instruction and a bitwise-and (`and`) instruction, respectively. In Fig. 7.1, the two consecutive instructions at *0x4010A9* establish the property: $edx_0 = 4 \times ecx + edx$, where edx_0 denotes the value contained in `edx` before the instructions are executed. This property is essential for inferring precise error triggers for the memory accesses at *0x4010B1* and *0x4010B7*. However, polyhedral analysis is not able to handle integer division with sufficient precision.

We overcome this problem by introducing additional auxiliary variables: each variable var that may hold a value for which both a quotient and remainder from division by k are computed is associated with two auxiliary variables, $var.quot_k$ and $var.rem_k$, which denote the quotient and the remainder, respectively. These variables represent the result of the corresponding integer computation symbolically. Furthermore, the following global constraint links the value of the variable to the values of the two auxiliary variables:

$$var = k \times var.quot_k + var.rem_k \ \wedge \ 0 \leq var.rem_k \leq k - 1. \tag{7.5}$$

We define a function *IntOp* : *Node* $\rightarrow \wp(\mathbb{N})$ that maps each node in the VDG to the corresponding set of divisors. To identify variables that may hold the values for which quotients and remainders are computed, we again use the value-dependence graph. Much like we did for allocation bounds, we compute the initial approximation for the *IntOp* function by including the mappings for the nodes that immediately participate in the corresponding integer operations. The *IntOp* function is then iteratively grown by adding mappings for nodes that are reachable or themselves reach the nodes for which the mappings have already been added to the function. There are multiple potential strategies for building the *IntOp* function, each with a corresponding precision/cost trade-off. Below we describe the two strategies that we experimented with:

Minimal construction [50]. The minimal-construction method looks for the VDG nodes that are reachable by backward traversals from both the division and remainder computation for the same divisor k (only affine-dependence edges are traversed). The auxiliary variables are associated with all of the nodes that are visited by the traversals, up to the first shared node: that is, for every such node u, the divisor k is added to the set *IntOp*(u).

For the above example, the starting point for the "quotient" traversal is the use of `ecx` at *0x4010AC*, and the starting point for the "remainder" traversal is the use of `edx` at *0x4010A9*: at these points, we generate assignments that directly use the corresponding auxiliary variables. The first shared node is the use of `edx` at *0x4010A7*: at that point, we generate numeric instructions that impose semantic constraints on the values of the auxiliary variables (see Fig. 7.2). This construction introduces a relatively small number of auxiliary variables, and allows polyhedral analysis to compute precise error triggers for the memory accesses at *0x4010B1* and *0x4010B7*.

Maximal construction. The alternative is to aggressively introduce auxiliary variables: that is, to associate the auxiliary variables with all the variables that are reachable by either backward or forward traversal of the VDG from the initial set of nodes. More formally, for each edge $u \to v$ in the VDG, the function *IntOp* is updated as follows: the divisors in *IntOp*(u) are added to the set *IntOp*(v), and vice versa. The process is repeated until the function *IntOp* stabilizes.

This approach introduces a large number of auxiliary variables, but also allows the analysis to handle pointer alignment. For instance, the overflow trigger that we obtain with this technique for the memory accesses at *0x401093* (see Fig. 7.1) looks as follows:

$$len \geq 4 \ \wedge \ 1 \leq ptr.rem_4 \leq 3 \ \wedge \ ptr.alloc_f \leq 4 - ptr.rem_4.$$

Note that this trigger is much more precise then the ones shown in §7.1.3: in particular, the constraint $1 \leq ptr.rem_4 \leq 3$ indicates that ptr must not be 4-byte aligned for the buffer overrun to occur; the constraint $ptr.alloc_f \leq 4 - ptr.rem_4$ is the strongest condition for the buffer overrun at *0x401093*.

Instruction translation. The numeric translations of x86 instructions must update the corresponding auxiliary variables in such a way that the global constraint shown in Eqn. (7.5) is satisfied for every annotated variable. This is not very hard in practice. The only complication is that the remainder auxiliary variables may wrap around as the result of an increment or a decrement. Thus, necessary checks must be inserted. In §7.3.7, we show an example translation.

7.3.6 Symbolic Memory Constants

The goal of our technique is to synthesize the summary of a library function by looking at its code in isolation. However, library functions operate in a larger context: they may access memory of the client program that was specified via their parameters, or they may access global structures that are internal to the library. The IR-recovery phase has no knowledge of either the contents or the structure of that memory: they are specific to the client application. As an example, from the IR-recovery perspective, `memset` parameter ptr may contain any memory address. Thus, from the point of view of numeric-program generation, a write into $*ptr$ may potentially overwrite any memory location: local and global variables, a return address on the stack, or even the code of the function. As the result, the generated numeric program, as well as the function summary derived from it, will be overly conservative (causing the client analysis to lose precision).

We attempt to generate more meaningful function summaries by using *symbolic memory constants* to model memory that cannot be confined to a specific a-loc by the IR-recovery phase. A unique symbolic memory constant is created for each unresolved memory access. From the numeric-analysis perspective, a symbolic constant is simply a global variable that has a special auxiliary variable $addr$ associated with it. This auxiliary variable represents the address of a memory location that the symbolic constant models. If the memory location may hold an address, then the corresponding symbolic memory constant has allocation bounds associated with it.

We illustrate the use of symbolic memory constants with an example that comes from function `_lseek`: The function `_lseek` moves a file pointer to a specified position within the file. It is declared as follows:

```
off_t _lseek(int fd, off_t offset, int origin);
```

fd is a file descriptor; *offset* specifies the new position of the pointer relative to either its current position, the beginning of the file, or the end of the file, based on *origin*.

A recurring memory-access pattern in `_lseek` is to read a pointer from a global table and then dereference it. Fig. 7.3 shows a portion of `_lseek` that contains a pair of such memory accesses: the first `mov` instruction reads the table entry, the second dereferences it. The registers `ecx` and `edx` hold the values *fd*$/32$ and *fd* **mod** 32, respectively. The global variable *uNumber* gives the upper bound for the possible values of *fd*. Symbolic constants mc_1 and mc_2 model the memory locations accessed by the first and second `mov` instructions, respectively.

Our technique synthesizes the following buffer-overrun trigger for the second `mov` instruction:

$$\text{0x424DE0} \le mc_1.addr \le \text{0x424DE0} + (\textit{uNumber} - 1)/8 \ \wedge \ mc_1.alloc_f <= 251$$

```
mov   eax, dword ptr [4×ecx + 0424DE0h]
```

assume$(mc_1.addr = \text{0x424DE0} + 4 * ecx);$

$eax \leftarrow mc_1;\ eax.alloc_f = mc_1.alloc_f;\ eax.alloc_b = mc_1.alloc_b;$

```
movsx ecx, byte ptr [eax + 8×edx + 4]
```

assert$(eax.alloc_f \leq 8 * edx + 5);$ **assert**$(eax.alloc_b + 8 * edx + 4 \geq 0);$

assume$(mc_2.addr = eax.alloc_b + 8 * edx + 4 \geq 0);\ ecx \leftarrow\ mc_2;$

Figure 7.3 Symbolic memory modeling: the symbolic constants mc_1 and mc_2 model the memory location accessed by the `mov` and `movsx` instructions, repsectively.

The above trigger can be interpreted as follows: *if any of the addresses stored in the table at 0x424DE0 point to a buffer of length that is less than 252 bytes, there is a possibility of a buffer-overrun error*. The error trigger is sufficient for a client analysis to implement sound error reporting: if the client analysis does not know the allocation bounds for pointers in the table at *0x424DE0*, it should emit an error report for this trigger at the call site to `_lseek`. However, we hope that the summary generated by our technique for the library-initialization code will capture the proper allocation bounds for the pointers in the table at *0x424DE0*. If that is the case, the analysis will not emit spurious error reports.

Note that an **assume** statement is used to restrict the value of variable $mc_1.addr$, rather than the assignment statement. The reason for that is that if we were to overwrite the address variable, we would not obtain the constraints on its value in the error triggers because the assignment kills the relationships between the value of $mc_1.addr$ at the call site of the function and the value of $mc_1.addr$ at the error point. Thus, we adopt the philosophy that before the memory access is reached by the analysis, the corresponding symbolic memory constant represents any memory location. The memory access “selects” a particular memory location that the symbolic memory constant must represent.

An interesting question is to see what soundness guarantees can be provided for the symbolic memory constants. An obvious concern is aliasing, i.e., what if two symbolic memory constants refer to the same location? Unfortunately, such aliasing can cause the translation to be unsound. Consider the following scenario: a write to a memory location is followed by a read from the same memory location, and the symbolic constant generated for the memory read is used in one of the error-trigger constraints. Suppose that the value stored at that memory location at the call site of the function does not satisfy the constraint; thus, the client analysis does not report an error. However, the write statement may have overridden the value that is stored in that memory location with the value that does violate the constraint, and the error is actually possible.

Currently, we do not have a good solution to this problem. One possibility is to perform a post-processing step: compare numerically the $addr$ variables associated with symbolic memory constants. Overlap between the values of two address variables indicates that the two symbolic memory constants may represent the same memory location, thus certain precautions must be taken before using the produced summary. For the same reason, the address variables associated with symbolic memory constants must be checked against the addresses of global variables that were resolved by the IR-recovery phase. Also, the client analysis must check that none of the symbolic constants represent memory locations in the activation record of the function to be invoked. Ultimately, we hope to add better memory modeling in the future.

7.3.7 Numeric-Program Generation

The actual numeric program generation is done by performing two passes over the x86 code: on the first pass, the value-dependence graph is constructed, and the set of variables that hold memory addresses (*Addr*) and the function that maps each node in the VDG to the corresponding set of divisors (*IntOp*) are built; on the second pass, the actual numeric program is generated.

Below, we illustrate the translation process by translating the instruction I: `inc` op_1. The overall translation is the concatenation of three pieces: basic translation, updates to the allocation bounds, and updates to the symbolic quotients and remainders. In the following, we use the function σ to map the VDG nodes to the corresponding numeric-program variables, i.e., $\sigma = \mathit{Var}_{\mathit{NP}} \circ \mathit{Var}_{\mathit{x86}}$.

Basic Translation. Basic translation is trivial: the increment instruction is translated into a numeric assignment statement that adds one to the numeric variable that represents the use of the operand op_1 and assigns the result to the numeric variable that represents the definition of op_1. The following numeric statement is produced:

$$\sigma(\mathit{Node}_D(I, op_1)) \leftarrow \sigma(\mathit{Node}_U(I, op_1)) + 1;$$

$\mathit{Node}_D(I, op_1)$	$\mathit{Node}_U(I, op_1)$	Generated Statements
$\notin$ *Addr*	–	nop;
$\in$ *Addr*	$\in$ *Addr*	$\sigma(\mathit{Node}_D(I, op_1)).alloc_f \leftarrow \sigma(\mathit{Node}_U(I, op_1)).alloc_f - 1;$ $\sigma(\mathit{Node}_D(I, op_1)).alloc_b \leftarrow \sigma(\mathit{Node}_U(I, op_1)).alloc_b + 1;$
$\in$ *Addr*	$\notin$ *Addr*	$\sigma(\mathit{Node}_D(I, op_1)).alloc_f \leftarrow ?;$ $\sigma(\mathit{Node}_D(I, op_1)).alloc_b \leftarrow ?;$

Table 7.1 The generation of updates for the allocation bounds.

Note that the same numeric variable will be used on both the left-hand-side and right-hand-side of the assignment statement.

Allocation-Bound Updates. The generation of numeric statements for maintaining allocation bounds makes use of the set *Addr*, which contains VDG nodes that are used to propagate memory addresses. If the node that corresponds to the definition of op_1 is not in the set, no statements need to be generated. Otherwise, the use of the operand op_1 is checked: if the node that corresponds to the use of the operand is also found in the set *Addr*, the allocation bounds for the definition of the operand are constructed from the allocation bounds associated with the use of the operand; on the other hand, if the node is not found, the allocation bounds for the definition of op_1 are handled conservatively. The details are shown in Tab. 7.1.

Quotient and Remainder Updates. The generation of numeric statements that maintain the symbolic quotients and remainders relies on the function *IntOp*. First, the set $\mathit{IntOp}(\mathit{Node}_D(I, op_1))$ is checked: if the set is empty, no code needs to be generated. Otherwise, for each divisor $k \in \mathit{IntOp}(\mathit{Node}_D(I, op_1))$, consistent updates for the auxiliary variables $quot_k$ and rem_k need to be generated. If $k \in \mathit{IntOp}(\mathit{Node}_U(I, op_1))$, then the quotient and the remainder associated with the use of the operand are used to compute the new values for the quotient and remainder associated with the definition of the operand (note that the numeric code must account for the possibility of wrap-around in the remainder value). Otherwise, conservative assumptions are made for the values of the quotient and remainder. The generation process is illustrated in Tab. 7.2.

for all $k \in$ *IntOp*$(\textit{Node}_D(I, op_1))$	Generated Statements
$k \in$ *IntOp*$(\textit{Node}_U(I, op_1))$	$\sigma(\textit{Node}_D(I, op_1)).quot_k \leftarrow \sigma(\textit{Node}_U(I, op_1)).quot_k;$ $\sigma(\textit{Node}_D(I, op_1)).rem_k \leftarrow \sigma(\textit{Node}_U(I, op_1)).rem_k + 1;$ **if**$(\sigma(\textit{Node}_D(I, op_1)).rem_k == k)\{$ $\quad\sigma(\textit{Node}_D(I, op_1)).rem_k \leftarrow 0;$ $\quad\sigma(\textit{Node}_D(I, op_1)).quot_k \leftarrow \sigma(\textit{Node}_U(I, op_1)).quot_k + 1;$ $\}$
$k \notin$ *IntOp*$(\textit{Node}_U(I, op_1))$	$\sigma(\textit{Node}_D(I, op_1)).quot_k \leftarrow ?;$ $\sigma(\textit{Node}_D(I, op_1)).rem_k \leftarrow ?;$ **assume**$(0 \leq \sigma(\textit{Node}_D(I, op_1)).rem_k \leq k - 1);$ **assume**$(\sigma(\textit{Node}_D(I, op_1)) ==$ $\qquad (k-1) \times \sigma(\textit{Node}_D(I, op_1)).quot_k + \sigma(\textit{Node}_D(I, op_1)).rem_k);$

Table 7.2 The generation of updates for the quotient and remainder auxiliary variables.

7.4 Numeric Analysis and Summary Generation

Our numeric analyzer is based on the Parma Polyhedral Library (PPL) and the WPDS++ library for weighted pushdown systems (WPDSs), and supports programs with multiple procedures, recursion, global and local variables, and parameter passing. The analysis of a WPDS yields, for each program point, a *weight*, or abstract state transformer, that describes how the program state is transformed on all the paths from the entry of the program to that program point. Linear-relation analysis [32] is encoded using weights that maintain two sets of variables: the *domain* describes the program state at the entry point; the *range* describes the program state at the destination point. The relationships between the variables are captured with linear inequalities. Given a weight computed for some program point, its projection onto the range variables approximates the set of states that are reachable at that program point. Similarly, its projection onto the set of domain variables approximates the precondition for reaching that program state.

7.4.1 Variable Packing

Function summaries are generated from the numeric-analysis results. In principle, summary transformers are constructed from the weights computed for the program points corresponding to procedure returns. Error triggers are constructed by back-projecting weights computed for the set of error program points. However, the poor scalability of polyhedral analysis is a major challenge: in practice, for most reasonable library functions, it will not be possible to analyze the generated numeric program in its entirety. We address this issue by breaking a large analysis problem into a set of smaller ones: we perform multiple analysis runs, each run still analyzes the entire program, but models only small subset of program variables. This technique is called *variable packing* (or, sometimes, *variable clustering*) [14, 87]. There is one difference in the way we use variable packing: the standard approach is to propagate all variable packs simultaneously, whereas we perform a separate analysis run for each pack. Note that, on the one hand, simultaneous pack propagation yields better analysis precision; but on the other hand, it puts much larger pressure on memory and is not parallelizable.

The main question that needs to be answered is how to identify variables that should go into the same pack. If we include too many variables, the analysis will not be efficient; if we include too few, the analysis will be imprecise. We generate packs with the use of the value-dependence graph: for each pack we identify a set of *generators*—that is, a set of variables (or rather, VDG nodes) that are of primary concern to us—and include into the pack all the nodes that are reachable from the generators by a backward traversal through VDG (*all* classes of dependence edges are used). One particular challenge that we face is that VDG graph contains only a single node per global or local variable. This approach worked well for identifying variables that must have auxiliary variables associated with them. However, in pack generation, this approach causes extra variables, which are of no use for the analysis, to be added to a pack. Intuitively, this happens because the treatment of variables in the VDG is control-flow insensitive: thus, some dependence chains in the VDG are not realizable in the actual program. We try to curb the propagation through the VDG with the use of heuristics: e.g., the backward traversal does not follow conditional edges out of the nodes that correspond to global variables. However, the heuristics are not yet mature enough to be reported.

In the next two sections, we describe pack generation for error triggers and summary transformers in more detail.

7.4.2 Error Triggers

A single pack is constructed for each safety-checked memory access; that is, the analysis run performed with this pack generates both the buffer-overflow trigger and the buffer-underflow trigger for the memory access. The set of generators for a memory-access pack contains the VDG nodes that are used directly in the memory-address computation. For the memory accesses performed by x86 instructions that execute hardware-supported loops, the node that indicates the number of loop iterations (i.e., the node that represents the use of register `ecx` at the corresponding instruction) is added to the set of generators.

For example, for the instruction I: `mov eax, [esi + 4 × ecx + 4]`, the set of generators contains *Node*$_U(I, esi)$ and *Node*$_U(I, ecx)$. For the instruction J: `rep movsd`, the following set is used:

$$\{\ \textit{Node}_U(J, esi),\ \textit{Node}_U(J, edi),\ \textit{Node}_U(J, ecx)\ \}$$

Splitting error-trigger formulas. The error triggers produced by our technique are represented by polyhedra. It may be of practical interest to decompose the error trigger into two parts:

- *Path component:* a polyhedron that corresponds to the precondition for reaching the memory access.
- *Error component:* a polyhedron that encompasses the general condition that has to be violated to cause an error.

The meet (or, in logical terms, conjunction) of the path component and the error component is equivalent to the original trigger.

For example, consider a buffer-overflow trigger from §7.3.5:

$$len \geq 4 \ \wedge \ 1 \leq ptr.rem_4 \leq 3 \ \wedge \ ptr.alloc_f \leq 4 - ptr.rem_4.$$

This trigger can be decomposed into a path component: $len \geq 4 \ \wedge \ 1 \leq ptr.rem_4 \leq 3$, which indicates that for the memory access to be reachable, the parameter len must have a value that is greater than 4, and the parameter ptr must not be aligned on a 4-byte word boundary; and an error component $ptr.alloc_f \leq 4 - ptr.rem_4$, which indicates that for the buffer overflow to happen at that memory location, the forward allocation bound must be less than the distance from ptr to the next 4-byte-aligned word.

We have a technique for automatically splitting error triggers into a path component and an error component. The technique operates as follows: the path component is trivially obtained by computing the precondition for the memory-access point. Observe that, because error points (to which control is diverted in the case of a memory error) are unique, each error trigger (i.e., a precondition for reaching an error point) is a subset of the corresponding path component. To obtain the error component, we need to find the most general polyhedron that, when intersected with the path component, yields the error trigger.

To compute error components, we defined a general operation on polyhedra: the operation takes two polyhedra P_1 and P_2, such that $P_1 \subseteq P_2$, and produces the most general polyhedron P, such that $P_2 \cap P = P_1$. The operation can be implemented by selecting only those constraints of the smaller polyhedron P_1 that are not satisfied by the larger polyhedron P_2. Interestingly enough, this operation is very close in spirit to the widening operator: in principle, the result of widening is constructed by selecting those constraints of the smaller polyhedron that are satisfied by the larger polyhedron[7].

[7] In practice, the implementation of widening is much trickier. See Chapter 2 for details. This may indicate that the implementation of the operation that we sketched above is not ideal, and can be improved by using ideas from the design of widening operators.

7.4.3 Summary Transformers

Generating summary transformers is somewhat harder than generating error triggers. First, as we said before, it is infeasible to generate a summary transformer with a single analysis run. Thus, we generate the summary transformer for each *target variable* of the library function. Target variables include the register `eax` at each return point, and the set of global variables that are updated within the function. For a particular target variable, the pack generator is a singleton set that contains that variable.

The summary transformer for a particular target variable is generated as follows: the weight computed for the return program point is transformed to forget the values of all “after” variables, except for the target variable. Also, information about local variables (if present) is dropped. Intuitively, the resulting polyhedron expresses the “after” value of the target variable solely in terms of “before” values of function parameters and global variables. To form the overall summary transformer, the transformers for the individual target variables are conjoined.

Note that this approach loses some precision: namely, the analysis does not capture numeric relationships among multiple target variables. That is, consider a function that on exit establishes the relationship $x + y = 5$, for some global variables x and y. Our technique will not be able to generate a summary transformer that captures this behavior, unless either x or y is preserved by the function (i.e., either $x' = x$ or $y' = y$).

Disjunctive Partitioning. Another challenge posed by generating summary transformers is due to non-distributivity of the polyhedral abstract domain. Library functions (as well as many non-library functions) typically have two kinds of paths: shorter “error” paths, which skip directly to the return statement if the corresponding error check evaluates to true, and longer “work” paths, which actually perform the function’s job. The majority of global-variable updates happen on the “work” paths, but not on the “error” paths. At the return point, the weights for the two kinds of paths are combined, which often causes precision to be lost: in relational polyhedral analysis, combining an identity transformation with any other transformation tends to lose precision. For instance, consider combining (i.e., joining) a polyhedron $\{x' = x\}$ with the polyhedron $\{x' = 5\}$. The resulting polyhedron includes the entire (x, x')-plane, i.e., all constraints on x and x' are lost.

To retain some precision, we resort to disjunctive partitioning: that is, we prevent certain weights from being combined. More precisely, we prevent the analysis from combining the weights from the paths on which the target variable is modified with the weight from the paths on which the target variable is preserved. (Recall, that due to packing, there is only one target variable per analysis run.) As a result, a pair of weights is computed for each program point, and thus a pair of summary transformers is generated for each target variable: the first transformer is typically an identity—the only interesting case is when this transformer is an annihilator (zero), which indicates that the target variable is updated on all paths through the function; the second transformer approximates all updates to the target variable that the function performs.

The above partitioning scheme is similar to the disjunctive partitioning performed by ESP [34]: that technique propagates functions that map states of an FSM (which is referred to as *property automaton*) to the elements of some abstract domain. At join points, only the elements that correspond to the same FSM state are joined together. Our technique uses a very simple "property" automaton that only has two states—the state "target variable preserved" and the state "target variable updated"—and a single transition that goes from "preserved" to "updated". The automaton starts in "preserved", and makes a transition to "updated" whenever a numeric assignment statement that updates the corresponding target variable is encountered by the analysis.

7.5 Experimental Evaluation

To experimentally evaluate the techniques presented in this chapter, we generated summaries for a number of functions in the standard C library. The particular library that we used in our experiments is the version of standard C library that is bundled with *Microsoft Visual Studio 6.0*: we used the version of the library for the *Release* build configuration (that is, the library code was optimized by the compiler).

We conducted two case studies. In the first study, we generated function summaries for the memory-manipulation library functions: *memset*, *memcmp*, and *memchr*.[8] The implementations

[8] Currently, we cannot handle the call *memcpy* because we cannot recover an accurate intermediate representation for it.

Library	x86	NP	Memory	Times (s)			Error Triggers	
Call	Instr.	Variables	Accesses	IR recovery	NP Generation	Analysis	Overrun	Underrun
memset	48	24	3	103	0.05	1.7	3/3	3/3
memchr	84	23	4	109	0.08	2.6	4/4	4/4
memcmp	86	36	12	100	0.16	13.4	8/12	12/12

Table 7.3 Analysis results for memory-manipulation library functions: the number of x86 instructions, the number of variables in the generated numeric program, and the number of safety-checked memory accesses are shown; times are given for IR recovery, numeric-program generation, and numeric analysis; the precision is reported as the number of triggers that are sufficiently precise to prevent the client analysis from generating spurious error reports.

of these library functions are relatively small: each consists of a single procedure; the number of x86 instructions ranges between 50 and 100. On the other hand, these library calls have fairly complex numeric behaviors: pointer arithmetic is used extensively, including checking for pointer alignment; the x86 instructions that execute hardware-supported loops and bitwise logical instructions are employed routinely. The goal of this study was to check whether our techniques provide sufficient precision to generate meaningful summaries for these library functions. We report the results of this study in §7.5.1.

The second study focused on stream library functions, such as *fclose*, *fopen*, *fflush*, etc. The implementations of these library calls are larger than the implementations of memory-manipulation calls: each implementation consists of several hundred instructions and multiple procedures. Also, internal library data structures (e.g., tables that store file information) are accessed and manipulated extensively. However, in contrast to the memory-manipulation functions, the numeric behavior of stream functions is quite simple. The goal of this study was to check the overall applicability and scalability of our techniques. The results of this study are reported in §7.5.2.

The experiments were conducted on two machines: the IR-recovery and numeric-program generation was done on a 1.83GHz Intel Core Duo T2400 with 1.5Gb of memory. The numeric analysis was done on a 2.4GHz Intel Pentium 4 with 4Gb of memory.

Run	Initial-state constraints	Analysis time (s)
1	$num \leq 3$	8.8
2	$num \geq 4,\ \ ptr\,\%\,4 = 0$	8.6
3	$num \geq 4,\ \ ptr\,\%\,4 \neq 0,\ \ ptr\,\%\,4 + num\,\%\,4 \leq 3$	12.1
4	$num \geq 4,\ \ ptr\,\%\,4 \neq 0,\ \ ptr\,\%\,4 + num\,\%\,4 = 4$	10.8
5	$num \geq 4,\ \ ptr\,\%\,4 \neq 0,\ \ ptr\,\%\,4 + num\,\%\,4 \geq 5$	10.5

Table 7.4 Pointer-alignment study for *memset*: five analysis runs were performed to generate error triggers that capture pointer alignment precisely; the initial-state constraints and the analysis time are shown for each run.

7.5.1 Case Study: Memory Functions

Tab. 7.3 shows the result of the application of our technique to the set of memory-manipulation library functions. For this set of experiments we used the *minimal construction* method for dealing with pointer-alignment checks (see §7.3.5). For each library function, intermediate-representation recovery took roughly a minute and a half, numeric-program generation was almost instantaneous, and numeric analysis took several seconds. We inspected the produced error triggers by hand: except for the four buffer-overrun triggers for *memcmp*, all the of the generated error triggers were sufficiently precise: they corresponded to the error triggers one would derive from the specification for those library calls. Implementation details, such as the fact that different paths are taken through the code depending on the alignment of the pointer arguments, were abstracted from the triggers.

The four triggers that were not captured precisely are due to the following loop in *memcmp* (we show a numeric-program excerpt, rather than the original x86 code):

```
if($(\mathbf{odd}(eax))$) $eax \leftarrow eax - 1$;
while($eax \neq 0$){
    $\ldots$
    $eax \leftarrow eax - 2$;
}
```

The if statement above makes sure that the value stored in register `eax` is even. The while loop decrements the value of `eax` by two on each iteration and uses a non-equality constraint to exit the loop: that is, this loop only terminates if the value in `eax` before the loop is even. The polyhedral abstract domain cannot represent the parity of a variable. Thus, the analysis presumes that there is no lower bound on the value of eax, which causes the error-triggers to be imprecise.

However, the technique that we introduced in §7.3.5 is able to deal with parity: if we introduce the auxiliary variables that symbolically represent "$eax \,\%\, 2$" and "$eax \,/\, 2$", and add the corresponding update statements for these variables to the numeric program, the analysis will obtain a precise bound for the value in register eax (to do that, the analysis must also track the parity of the parameter that specifies the length of the two buffers that *memcmp* compares). We manually instrumented the numeric program and checked that the triggers generated from that program are indeed precise. However, we have not yet implemented an automatic way for detecting these cases and adding necessary instrumentation to the program.

To experimentally evaluate the *maximal construction* method from §7.3.5, we applied it to the *memset* library call. The generated numeric program had an increased number of variables compared to the minimal construction method (31 instead of 24). Also, numeric analysis applied directly to the problem took several hours and yielded imprecise results. We traced the problem to the non-distributivity of the polyhedral abstract domain: the relationships between symbolic remainders and quotients are generally non-convex—approximating those relationships caused complex polyhedra (i.e., polyhedra with large numbers of vertices and constraints) to arise in the course of the analysis. Complex polyhedra, in turn, caused the analysis to be slow and imprecise. To prevent the joins of "incompatible" polyhedra, we manually partitioned the paths that the analysis explores. The paths were partitioned by imposing a set of constraints on the program states at the entry of the library call and performing multiple analysis runs to cover the entire set of initial states. Tab. 7.4 shows the results we obtained: 5 analysis runs were required, each run took on the order of 10 seconds, and the triggers that were generated gave the most precise error conditions that can be inferred for this implementation of *memset*. However, the question of automating such paths partitioning remains to be addressed in the future.

Library Call	x86 Instr.	Proc. Count	Numeric Program Variables global	local (max)	local (avg)	Memory Accesses	IR Recovery Time (s)	NP Generation Time (s)
fclose	843	12	336	37	14	115	52.9	2.8
fflush	443	8	156	30	11	38	43.4	1.2
fgets	469	7	216	30	14	75	45.0	1.2
fopen	1419	18	258	35	15	107	69.5	4.0
fputs	784	13	266	40	15	98	59.5	2.7
fread	524	7	230	30	15	81	49.2	1.6
fseek	514	7	192	30	15	59	46.6	1.3
ftell	249	4	138	27	13	30	40.7	0.6
fwrite	600	9	238	40	16	75	57.2	2.1

Table 7.5 Numeric-program generation for stream library functions; the number of x86 instructions and the number of procedures in the implementation of each library function is shown; for the numeric program, the number of global variables, and the maximum and the average number of local variables are shown. Last two columns give the times spent on recovering intermediate representations and on generating numeric programs for each library function.

7.5.2 Case Study: Stream Functions

Tab. 7.5 shows the stream library functions for which we generated summary functions. Of particular interest is the number of variables in the numeric program that our technique generates. The maximum number of variables that the numeric analyzer must track at any point throughout the analysis is given by the sum of the global variables and the maximum number of local variables among all of the procedures in the generated program.[9] In the end, the overall number of variables that the analysis needs to track measures in the hundreds, and exceeds the number that polyhedral analysis is able to handle in practice. In fact, if we feed any of these numeric programs to the numeric analyzer, the analyzer runs out of memory while constructing the pushdown system—that is, even before the iterative computation begins. To analyze these numeric programs, we rely on the variable-packing technique, which we described in §7.4.1.

Tab. 7.5 also reports the time spent for recovering intermediate representations and for generation of numeric programs. The IR-recovery phase took approximately one minute for each library function. We believe that this can be improved by better engineering of the tool—something that we have not yet addressed. For instance, we can use a single CodeSurfer run to recover IRs for

[9] In fact, in addition to these, a few extra variables may be required to support parameter passing.

Library	Pack	Variables/Pack		Analysis Precision			Analysis Time	
Call	Count	Maximum	Average	Both	One	None	Total (min)	Average/Pack (s)
fclose	115	63	23	64	7	44	79.0	41.2
fflush	38	35	21	22	2	14	10.0	15.8
fgets	75	31	12	67	7	1	5.5	4.4
fopen	107	37	17	68	2	37	46.5	26.1
fputs	98	69	13	78	3	17	24.5	15.0
fread	81	46	14	70	0	11	13.8	10.2
fseek	59	30	15	56	1	2	7.8	7.9
ftell	30	29	18	30	0	0	2.3	4.5
fwrite	75	69	16	69	0	6	20.0	16.0

Table 7.6 Error-trigger generation for stream library functions; the *pack count* indicates the number of analysis runs performed: each analysis run generates a buffer-overflow trigger and a buffer-underflow trigger; *Variable count:* the number of variables that the analysis needs to model simultaneously (i.e., globals + locals), the *maximum* number across all packs and the *average* number per pack is shown; *Analysis Precision:* measured as the number of packs for which *both*, *one*, or *none* of the generated triggers are meaningful.

all library functions, as opposed to performing a single CodeSurfer run for *each* library function as is done now. Numeric-program generation takes only a few seconds per library function; thus, its performance is not of immediate concern. However, we believe that the efficiency of numeric-program generation can also be improved.

Error Triggers. To generate error triggers, we used variable-packing approach presented in §7.4.2: that is, a separate variable pack was generated for each memory access that is checked for safety, and a separate analysis run was performed for each pack. Tab. 7.6 shows the results obtained. Note that the number of variables in each pack is much more manageable compared to the entire program, and on average, is rather small (under 25 variables). On average, an analysis run (for a single variable-pack) takes under a minute. The overall analysis takes on the order of few dozen minutes per library function.

An interesting question is how to judge the quality of the error triggers obtained. If we had an implementation of memory-safety analysis that could use the produced triggers, we could have measured the effectiveness of our technique by reduction/increase in the number of false positives reported by the analysis. However, currently, there is no client analysis that can use the summaries

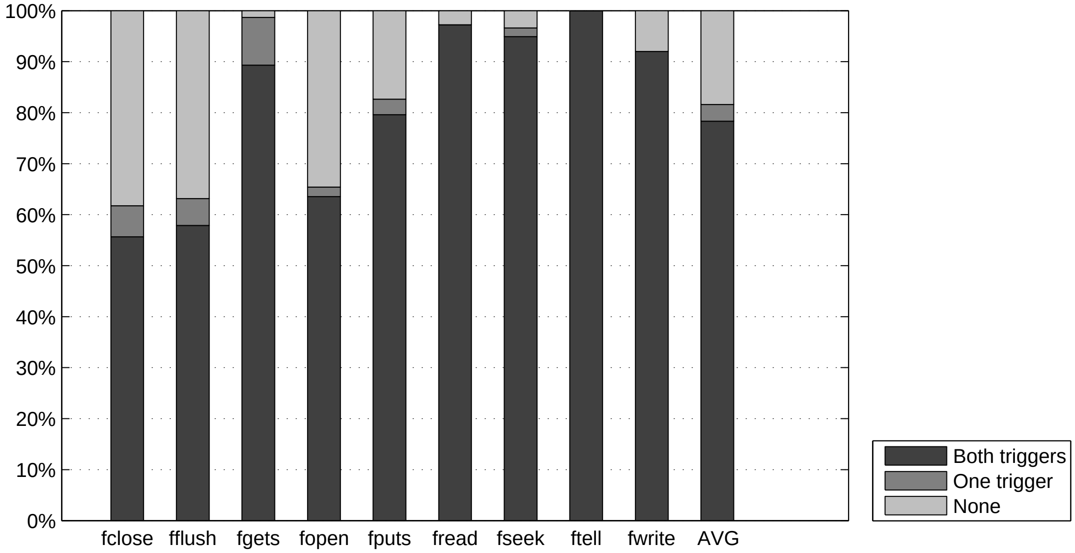


Figure 7.4 Error triggers for stream library calls: the percentages of memory accesses for which both generated triggers are meaningful, one of the generated triggers is meaningful, and none of the generated triggers are meaningful, are shown.

that we produce. Also, the number of error triggers generated by our technique is sufficiently large to prevent us from inspecting each trigger manually.

We used the following automatic technique to assess the quality of the produced triggers: each trigger is split into two parts—the path component and the error component—as described in §7.4.2. By construction, error components may only contain constraints that are relevant to the error. Therefore, our automatic trigger-assessment technique is based on inspecting the error components of the generated triggers.

We declare an error trigger to be *meaningful* if its error component is *not* $\top$ (or *true*, in logical terms). The error trigger $\top$ indicates that the corresponding error may happen during the invocation of the library call regardless of the program state in which the call was invoked. Generally, we do not expect library functions to behave in this way; thus, $\top$ most likely indicates that the analysis has lost precision.

We eliminated the path component of the trigger from consideration because the path component may contain some constraints that are not relevant to the error (e.g., some program invariant), thus making the overall trigger non-$\top$.

Fig. 7.4 shows the results of our error-trigger-quality assessment. For each library function, we report the percentages of memory accesses for which both generated triggers (i.e., the buffer-overrun trigger and buffer-underrun trigger) are meaningful; only one of the generated triggers (i.e., either the buffer-overrun trigger or buffer-underrun trigger) is meaningful; and neither of the two generated triggers is meaningful. As the rightmost column indicates, our technique is able to generate meaningful error triggers for about 80% of memory accesses.

The call *fclose* seems to provide the biggest challenge for our technique: the numeric analysis takes the longest on *fclose* (79 minutes) and yields the poorest precision (for only 56% of memory accesses are both generated error triggers meaningful). We took a detailed look at the 15 longest analysis runs for *fclose* (out of 115): the cumulative time for those analysis runs accounted for more than 50% of the overall analysis time (40 out of 79 minutes); also, meaningful triggers were generated by only 3 of those analysis runs (out of 15). Similar situations occur for other library calls too: a small number of analysis runs takes an increasingly long time and produces poor results, whereas the remaining analysis runs are fast and yield good precision. Our experience indicates that the likely cause for this behavior is the failure of our current techniques for variable-pack identification to produce reasonable variable packs for the corresponding memory accesses. One future direction for this work is to improve the pack-identification techniques: we believe that better variable-pack identification will significantly improve both the precision and the efficiency of the analysis.

Summary Transformers. Initially, we focused on the generation of error triggers because they are somewhat easier to generate and their quality is easier to assess. Recently, we switched our attention to the generation of summary transformers. Our initial approach was to apply directly the techniques that we designed for error-trigger generation; the only exception was the technique for variable-pack identification—we use the technique described §7.4.3 to identify variable packs for

Library	Pack	Variables/Pack		Transformer Precision			Analysis Time	
Call	Count	Maximum	Average	Full	Partial	Bad	Total (min)	Average/Pack (s)
fclose	43	67	26	8	1	34	115.5	161.2
fflush	28	30	14	6	0	22	16.1	34.5
fgets	21	35	11	10	8	3	20.9	59.6
fopen	22	40	25	6	0	15	64.5	176.0
fputs	36	73	10	14	8	14	57.3	95.5
fread	23	50	14	2	1	20	414.9	1082.3
fseek	11	22	9	5	4	2	0.8	4.1
ftell	4	46	16	1	1	2	1.3	18.8
fwrite	25	73	13	12	11	2	49.1	117.9

Table 7.7 Summary-transformer generation for stream library functions; the *pack count* indicates the number of analysis runs performed: each analysis run generates a transformer for a single target variable; *Variable count:* the number of variables that the analysis needs to model simultaneously (i.e., globals + locals)—the *maximum* number across all packs and the *average* number per pack are shown; *Analysis Precision:* the numbers of target variables for which *full* transformers, *partial* transformers, and *bad* transformers are obtained (for definitions, see §7.5.2).

generating summary transformers. The initial results were very poor: the technique was not able to produce any meaningful summary transformers.

The primary reason for the failure of the analysis, as we discussed in §7.4.3, was the precision loss due to non-distributivity of the polyhedral abstract domain: the combination of the weights computed for the paths on which the target variable was updated with the weights for the paths on which the target variable was preserved cannot be represented precisely with a single polyhedron. To overcome this problem, we modified the analysis to compute two weights: one represents the program-state transformation for the paths that modify the target variable, the other represents the program-state transformation for the paths that preserve the target variable. This modification allowed the analysis to produce much more meaningful summary transformers.

Tab. 7.7 shows the results that we obtained. First, note that the variable packs are somewhat larger than the ones identified for error-trigger generation. Also, the analysis takes significantly longer to complete. To estimate the precision of the generated summary transformers, we manually classified the resulting transformers into three categories. The target variables are the variables that are updated by the library call: they include the return value of the call, the global variables that are explicitly assigned to, and symbolic memory constants that model unresolved memory writes.

The latter account for the majority of target variables. Two particular things that are of interest in a summary transformer for a particular target variable are (i) what value is assigned to the target variable, and (ii) whether we can precisely identify what memory location the target variable represents. Item (ii) above is only of relevance for the symbolic memory constants. We recognize the following categories of summary transformers:

- **Full Transformers:** These transformers capture both the resulting value and the address (or, possibly, the range of addresses) for the corresponding target variable. The addresses of target variables are of more concern to us because we have to link symbolic memory constants to the corresponding memory locations. Our treatment of the resulting values is less strict: that is, we declare the value to be "captured" if any constraints on the value were inferred.

- **Partial Transformers:** These transformers capture the address (or the range of addresses) for the target variable, but lose the information about the resulting value; that is, these transformers represent definite or conditional kills of the corresponding target variable. Only the transformers for symbolic memory constants are classified as partial—partial transformers for global variables are not very interesting; that is, there are much less expensive ways of obtaining information about potential modifications of global variables, such as GMOD analysis [25].

- **Bad Transformers:** These transformers do not capture either the value or the address of the target variable. If the target variable is a return value or a global variable, a bad transformer corresponds to a kill (definite or conditional) of that variable; if the target variable is a symbolic memory constant, a bad transformer indicates that any memory location could have potentially been updated—this is the worst possible scenario for the client analysis.

The chart in Fig. 7.5 shows the percentages of target variables for which full, partial, and bad transformers were inferred. On average, full transformers are inferred for about 30% of target variables; partial transformers are inferred for another 15%, leaving more than half of target variables with bad transformers. These results, however, are preliminary, and are included in this thesis

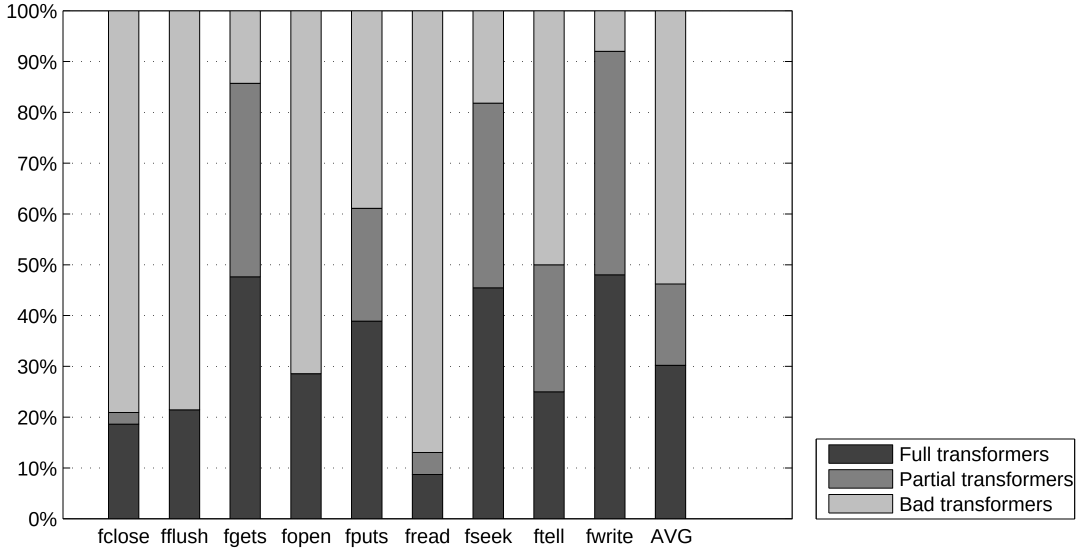


Figure 7.5 Summary transformers for stream library calls: the percentages of target variables for which *full* transformers, *partial* transformers, and *bad* transformers are obtained (for definitions, see §7.5.2).

primarily for the sake of completeness. In the generation of error triggers, the main cause of the analysis imprecision was poor variable-pack identification. The above results may indicate that the variable-pack-identification techniques that worked satisfactorily for error-trigger generation, do not work that well for the generation of summary transformers. Thus, in the future, better identification techniques that are tuned to summary-transformer generation need to be designed.

7.6 Related Work

Summary functions have a long history, which goes back to the seminal work by Cousot and Halbwachs on linear-relation analysis [32] and the papers on interprocedural analysis of Cousot and Cousot [30] and Sharir and Pnueli [106]. Other work on analyses based on summary functions includes [11, 71, 94], as well as methods for pushdown systems [16, 17, 41, 97], where summary functions arise as one by-product of an analysis.

A substantial amount of work has been done to create summary functions for alias analysis or points-to analysis [22, 58, 77, 98, 115], or for other simple analyses, such as lock state [116]. Those algorithms are specialized for particular problems; more comprehensive approaches include the work on analysis of program fragments [99], componential set-based analysis [42], and use of SAT procedures [116].

The relevant-context-inference algorithm of Chatterjee et al. [22] determines points-to information for a subset of C++. It works bottom-up over the call graph, analyzing each method using unknown initial values for parameters and globals. The goal is to obtain a summary function together with conditions on the unknown initial values.

The work on points-to and side-effect analyses for programs built with precompiled libraries [98] concerned flow-insensitive and context-insensitive analyses. Such analyses ignore the order among program points in procedures and merge the information obtained for different calling contexts.

Some of the work cited above explicitly mentions separately compiled libraries as one of the motivations for the work. Although the techniques described in the afore-mentioned papers are language-independent, all of the implementations described are for source-code analysis.

Guo et al. [55] developed a system for performing pointer analysis on a low-level intermediate representation. The algorithm is only partially flow-sensitive: it tracks registers in a flow-sensitive manner, but treats memory locations in a flow-insensitive manner. The algorithm uses partial transfer functions [115] to achieve context-sensitivity, where the transfer functions are parameterized by “unknown initial values”.

Kruegel et al. [72] developed a system for automating mimicry attacks. (i.e., attacks that evade detection by intrusion detection systems that monitor sequences of system calls). Their tool uses symbolic-execution techniques on x86 binaries to discover attacks that can give up and regain execution control by modifying the contents of the data, heap, or stack so that the application is forced to return control to injected attack code at some point after a system call has been performed. Cova et al. [33] used this platform to apply static analysis to the problem of detecting security vulnerabilities in x86 executables. In both of these systems, alias information is not available.

In our work, we make use of a-locs (variable proxies), alias information, and other IRs that have been recovered by the algorithms used in CodeSurfer/x86 [8, 9]. The recovered IRs are used as a platform on which we implemented a relational analysis that synthesizes summary functions for procedures.

Chapter 8

Conclusions and Future Work

In this thesis, we presented a collection of techniques for enhancing the precision and applicability of numeric program analysis. The proposed techniques are orthogonal to each other and can be (and, in fact, should be) combined and used together in the implementations of numeric-program-analysis tools. The techniques are not specific to any particular numeric abstraction or any particular iterative-computation-based analysis engine: rather, certain minimal requirements are placed (in the form of an interface) on these components. For the numeric abstractions, the techniques adhere to the interface imposed by the abstract-interpretation framework: i.e., abstractions are viewed as domains (partially-ordered sets) that provide certain operations; e.g., meet, join, widening, etc. As a result, our techniques can be instantiated with any existing numeric abstraction or with any numeric abstractions to be introduced in the future, as long as those abstraction adhere to the required interface.

For program analyzers, the guided-static-analysis technique from Chapter 5 imposes a general interface that is inspired by model checking and by transition systems: the only assumption that is placed on the analyzer is that it soundly approximates a set of program states that can be reached by some execution of a program from a specified set of initial states. This interface is sufficiently generic to allow guided static analysis to be easily integrated into a wide range of existing analyzers. We integrated guided static analysis into the existing library for weighted pushdown systems, WPDS++ [69] with minimal effort: no changes were required to the implementation of the fix-point-computation engine of the library.

In the following, we briefly summarize each of the proposed numeric-program-analysis techniques, and indicate directions for future work.

Summarizing Abstractions. Chapter 3 addressed the question of how to represent numeric states of the systems where the number of numeric objects that the analysis must keep track of varies from state to state and is, in general, unbounded. The chapter shows how to systematically construct summarizing abstract domains, which are capable of representing universal properties of unbounded collections objects, from existing abstract domains. Summarizing abstract domains can be used in conjunction with some form of a summarizing abstraction to represent numeric states for systems that manipulate unbounded numbers of objects (e.g., for programs that perform dynamic-memory allocation). The requirements that are placed on the summarizing abstraction are minimal: summarization can be as simple as collapsing together all memory locations created at the same allocation site, or as complex as canonical, which is abstraction used in state-of-the-art shape-analysis tools [78, 100].

The key difference between summarizing abstraction and other techniques that are used to represent universal properties for unbounded numbers of objects is that summarizing abstractions model all objects in the system as first-class citizens: that is, the properties of summarized objects are synthesized and represented in the same way as the properties of non-summarized objects. Other techniques, typically, rely on special representations for the universal properties, such as *parametrized* predicates [110] and *range* predicates [67]; such approaches require the design of special techniques and heuristics to reflect the effect of program statements on such predicates. In contrast, Chapter 3 showed how to create sound transformers uniformly for all summarizing abstractions.

Future Directions. There are a number of interesting future directions for this work:

- *Predicate abstraction.* Predicate abstraction is a very popular technique in software verification: it is a main ingredient in *parsimonious* abstractions, which are viewed as one of the keys to future scalability of software verification. Predicate abstraction is, in fact, an abstract domain that can (i) capture correlations between numeric and Boolean values, and (ii) capture disjunctions (and, consequently, implications) of numeric properties. Item (ii) above is of particular interest for summarizing abstractions because it would significantly extend the class of properties that can be automatically captured (see §3.5).

The summarizing extension for predicate abstraction can be trivially constructed based on the material in Chapter 3. An interesting feature of such an extension is that the predicates that are used to instantiate it are implicitly universally quantified. However, note that the techniques in §3.4 allow to transform the values of such predicates with the use of decision procedures that do not support quantification. On the other hand, existing iterative-refinement techniques cannot be used directly to derive such universally-quantified predicates. An interesting research direction is to investigate the use of summarizing abstractions in the context of predicate abstraction.

- *Aggregation functions.* The first two steps of summarizing abstraction (referred to as *partial* abstraction in Chapter 3) *aggregate* the values associated with the objects that are summarized together. The particular aggregation that we explored is collecting the values into a set. However, one can imagine other aggregation functions that could have been used: e.g., selecting a minimal or a maximal value, computing the sum or the average of the values, etc. An interesting research question is whether the techniques in Chapter 3 can be generalized to be applicable to arbitrary aggregation functions. Also, would be interesting to see if there are any applications in the area of program analysis that may require such aggregation functions.

Analysis of Array Operations. Chapter 4 presented a framework for analyzing code that manipulates arrays, e.g., sorting routines, initialization loops, etc. In particular, the target of the analysis was to infer universal properties of array elements. The framework combines two techniques: canonical abstraction from the realm of shape analysis, and the summarizing numeric abstractions presented in Chapter 3. Canonical abstraction was used to summarize together contiguous segments of array elements, at the same time leaving the elements that are indexed by loop-induction variables as non-summary elements (to facilitate strong updates in the body of the loop). Summarizing abstractions were used to keep track of values and indices of the array elements. We used a prototype implementation of the analysis to successfully analyze a number of array-manipulation routines, including partial array initialization and an implementation of an insertion-sort routine.

Future Directions. In its current state, the array-analysis technique is suitable for analyzing small, single-procedure programs that encompass the essence of array operations. Many issues must still be addressed before the technique can be applied to “real-world” programs. In particular, the array-partitioning has to be confined to small regions of the program where it is absolutely necessary, otherwise the analysis will not scale. Techniques that automatically identify those regions would have to be developed. Also, for the cases where summarizing abstractions fall short, automatic techniques for inferring necessary auxiliary predicates must be designed.

Guided Static Analysis. Chapter 5 presented the framework of guided static analysis, a technique for controlling the exploration performed by an analysis of the space of program states. The exploration of the state space is guided by deriving a sequence of program restrictions: each restriction is a modified version of the original program that only contains a subset of behaviors of the original program; standard analysis techniques are used to analyze the individual restrictions in the sequence.

The instantiations of guided-static-analysis framework were used to improve the precision of widening. Widening precision is essential to the overall precision of numeric analysis. A number of ad-hoc techniques for improving widening precision has been proposed since the introduction of widening in the 1970s. We believe that the techniques proposed in §5.4 and §5.6 are among the most systematic techniques for improving widening precision to date.

Of particular interest is the *lookahead-widening* technique (§5.6). Lookahead widening can be easily integrated into existing analysis tools: all it takes is a simple extension to the abstraction that is currently used by an analyzer; no changes to the analysis engine are required. We integrated lookahead widening into two numeric analyzers: (i) a small intra-procedural analyzer constructed according to the principles of Chapter 2, and (ii) an analyzer based on an of-the-shelf WPDS library. In both cases, the integration required minimal effort, and the precision of both analyzers was substantially improved (see §5.7).

Future directions. In this thesis, we investigated the use of guided static analysis only in the context of improving widening precision. Also, the construction of program restrictions by the proposed instantiations of the framework was done by completely removing certain edges from the control-flow graph of the program. Note, however, that the framework allows for more fine-grained ways to restrict program behaviors: in particular, it allows to strengthen the transformers associated with the edges of a CFG. An interesting research direction would be to find other applications for guided static analysis that may exercise the capabilities of the framework to a greater degree. As an example, consider a program that performs a weak update:[1] a restriction for that program can be derived by replacing the weak update either by the corresponding strong update, or by an identity transformation. However, it is not clear whether this approach will result in any precision gain.

Numeric Analysis and Weighted Pushdown Systems. Chapter 6 investigated the use weighted-pushdown-system machinery as an engine for numeric analysis. Our main numeric-analysis tool, which we used to conduct the experiments described in Chapters 5 and 7, is built on top of an off-the-shelf weighted-pushdown-system library, WPDS++ [69]. In essence, the tool implements a version of relational polyhedral analysis [30, 32, 66] in the framework of weighted pushdown systems. The key advantage of a pushdown-system-based implementation of the analysis is the ability to answer stack-qualified queries: that is, the ability to determine the properties that arise at a program point in a specified set of calling contexts.

Future Directions. The precision of relational polyhedral analysis is severely affected by the non-distributivity of the polyhedral abstract domain. As we pointed out in §7.5, in some cases, combining "incompatible" weights significantly slows down the analysis, and, simultaneously, leads to a loss of precision. Currently, most research in the area of weighted pushdown systems relies on the assumption that the weights are distributive. In the future, the issue of improving precision for non-distributive weight domains will have to be addressed, possibly by leveraging some of the existing techniques for disjunctive partitioning [5, 34, 83, 102].

Library Analysis and Summarization. Chapter 7 addressed the question of automatic generation of summaries for library functions from the low-level implementation of the library (i.e., from the library's binary). Currently, library functions (for which source code is rarely available), pose a major stumbling block for source-level program-analysis tools. Typically, models for library

[1] Weak updates were discussed in Chapters 3 and 4.

functions are manually constructed by either hardcoding them into the analyzer, or by providing a collection of hand-written function stubs that emulate certain aspects of the library. Automatic construction of function summaries eliminates the lengthy and error-prone manual construction of models for library-functions. Additionally, because summaries are generated directly from the library implementation, the resulting summaries automatically account for the deviations of that particular implementation of the library from the library's general specification.

In Chapter 7, we took the first steps towards automatic construction of summaries for library functions. We selected one particular client analysis—*memory safety analysis*, which was known to be reducible to numeric program analysis [37, 38]—and used our expertise, as well as some of the numeric-program-analysis techniques described in this thesis, to build a tool for the automatic construction of function summaries suitable for memory-safety analysis. The results in §7.5 indicate that the approach we have taken—that is, translating x86 code into a numeric program and analyzing the resulting numeric program with relational polyhedral analysis—is feasible in practice. However, in its current state, our tool is still far from being useful in practice.

Future Directions. Aside from immediate needs that have to be addressed, such as the design of better variable-pack-identification techniques, and getting a better grip on disjunctive partitioning, there are two issues that pose interesting research questions:

- *Better memory modeling.* As we suggested in §7.3.6, the use of symbolic memory constants may compromise the soundness of the analysis. However, the analysis captures (numerically) the addresses of the memory locations that the symbolic memory constants represent: a post-processing step may inspect the possible values of $addr$ variables to detect possible aliasing among symbolic memory constants, and discard function summaries that are affected by such aliasing. A interesting research question is how to design a better scheme for modeling unresolved memory. Ideally, such a scheme would integrate well into the WPDS paradigm. One possibility is to incrementally refine the memory model, i.e., use the values of $addr$ variables, and possibly the values of some new auxiliary variables, to refine the memory model (e.g., replace all symbolic memory constants that refer to the same location by a single global variable), and then rerun the analysis.

- *Consolidation of error triggers.* The current implementation of the tool generates an error trigger for each memory access. Furthermore, due to variable packing, correlations between individual triggers are typically lost. As a result, the error conditions that the tool produces are largely redundant, whereas in principle they could be reduced to a small number of fairly general error conditions. Techniques that are able to perform such consolidation would be desirable.

Bibliography

[1] X. Allamigeon, W. Godard, and C. Hymans. Static analysis of string manipulations in critical embedded C programs. In *Static Analysis Symp.*, pages 31–51, 2006.

[2] A. Armando, M. Benerecetti, and J. Mantovani. Abstracting linear programs with arrays into linear programs. Tech. rep., AI Lab. Univ. of Genova, April 2005.

[3] A. Armando, M. Benerecetti, and J. Mantovani. Model checking linear programs with arrays. *Electr. Notes Theor. Comput. Sci.*, 144(3):79–94, 2006.

[4] A. Armando, M. Benerecetti, and J. Mantovani. Abstraction refinement of linear programs with arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 373–388, 2007.

[5] R. Bagnara, P. Hill, and E. Zaffanella. Widening operators for powerset domains. In *Verification, Model Checking, and Abstract Interpretation*, pages 135–148, 2004.

[6] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Static Analysis Symp.*, volume 2694, pages 337–354, 2003.

[7] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *Static Analysis Symp.*, volume 2477, pages 213–229, 2002.

[8] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Int. Conf. on Comp. Construct.*, pages 5–23. Springer-Verlag, March 2004.

[9] G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *Verification, Model Checking, and Abstract Interpretation*, 2007.

[10] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *Proc. IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, October 2005.

[11] T. Ball and S.K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 97–103, 2001.

[12] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Found. of Comput. Sci.*, 14(4):605–624, 2003.

[13] C. Bartzis and T. Bultan. Widening arithmetic automata. In *Int. Conf. on Computer Aided Verification*, pages 321–333, 2004.

[14] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation.*, pages 85–108. Springer-Verlag, 2002.

[15] I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *Int. Conf. on Computer Aided Verification*, 2007.

[16] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Int. Conf. on Concurrency Theory*, pages 135–150, 1997.

[17] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Symp. on Princ. of Prog. Lang.*, pages 62–73, 2003.

[18] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, pages 128–141, 1993.

[19] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically protecting against integer-based vulnerabilities. In *Symp. on Network and Distributed Systems Security*, 2007.

[20] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999.

[21] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 334–349, 2006.

[22] R. Chatterjee, B.G. Ryder, and W. Landi. Relevant context inference. In *Symp. on Princ. of Prog. Lang.*, pages 133–146, 1999.

[23] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *Static Analysis Symp.*, pages 312–327, 2004.

[24] R. Clarisó and J. Cortadella. The octahedron abstract domain. *Sci. of Comput. Program.*, 64(1):115–139, 2007.

[25] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Symp. on Princ. of Prog. Lang.*, pages 49–59, 1989.

[26] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Int. Conf. on Computer Aided Verification*, pages 462–475, 2005.

[27] P. Cousot. Verification by abstract interpretation. In N. Dershowitz, editor, *Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, pages 243–268, Taormina, Italy, 2003.

[28] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd. Int. Symp on Programming*, Paris, April 1976.

[29] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, 1977.

[30] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Descriptions of Programming Concepts*. North-Holland, 1978.

[31] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, 1979.

[32] P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Symp. on Princ. of Prog. Lang.*, 1978.

[33] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *Annual Computer Security Appl. Conf.*, 2006.

[34] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Conf. on Prog. Lang. Design and Impl.*, pages 57–68, 2002.

[35] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Conf. on Prog. Lang. Design and Impl.*, pages 230–241, 1994.

[36] A. DiGiorgio. The smart ship is not the answer. *Naval Institute Proceedings Magazine*, June 1998.

[37] N. Dor, M. Rodeh, and S. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis Symp.*, pages 194–212, 2001.

[38] N. Dor, M. Rodeh, and S. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Conf. on Prog. Lang. Design and Impl.*, pages 155–167, 2003.

[39] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Int. Conf. on Computer Aided Verification*, pages 232–247, 2000.

[40] J. L. Lions et al. ARIANE 5, Flight 501 failure, Report by the inquiry board. Available at http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, 1996.

[41] A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electr. Notes Theor. Comput. Sci.*, 9, 1997.

[42] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Conf. on Prog. Lang. Design and Impl.*, pages 235–248, 1997.

[43] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Symp. on Princ. of Prog. Lang.*, pages 191–202, 2002.

[44] S. Gaubert, E. Goubault, A. Taly, and S. Zennou. Static analysis by policy interation on relational domains. In *European Symp. on Programming*, 2007.

[45] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Conf. on Prog. Lang. Design and Impl.*, pages 213–223, 2005.

[46] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *Static Analysis Symp.*, pages 144–160, 2006.

[47] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529, 2004.

[48] D. Gopan and T. Reps. Lookahead widening. In *Int. Conf. on Computer Aided Verification*, pages 452–466, 2006.

[49] D. Gopan and T. Reps. Guided static analysis. In *Static Analysis Symp.*, 2007. To appear.

[50] D. Gopan and T. Reps. Low-level library analysis and summarization. In *Int. Conf. on Computer Aided Verification*, pages 68–81, 2007.

[51] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Symp. on Princ. of Prog. Lang.*, pages 338–350, 2005.

[52] P. Granger. *Analyses Semantiques de Congruence*. PhD thesis, Ecole Polytechnique, 1991.

[53] P. Granger. Static analyses of congruence properties on rational numbers. In *Static Analysis Symp.*, pages 278–292, 1997.

[54] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Symp. on Princ. of Prog. Lang.*, pages 122–131, 2005.

[55] B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *Int. Symp. on Code Generation and Optimization*, pages 291–302, 2005.

[56] N. Halbwachs. On the design of widening operators. Invited tutorial for Verification, Model Checking, and Abstract Interpretation, January 2006.

[57] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[58] M.J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *Trans. on Software Engineering*, 22(7), 1996.

[59] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symp. on Princ. of Prog. Lang.*, pages 58–70, 2002.

[60] CERT: Computer Emergency Readiness Team. http://www.cert.org/.

[61] IDAPro disassembler, http://www.datarescue.com/idabase/.

[62] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Int. Conf. on Computer Aided Verification*, pages 137–151, 2006.

[63] B. Jeannet. The convex polyhedra library New Polka. Available online at http://pop-art.inrialpes.fr/people/bjeannet/newpolka/.

[64] B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *Int. Workshop on Numerical and Symbolic Abstract Domains.*, 2005.

[65] B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *Static Analysis Symp.*, pages 186–202, 2005.

[66] B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Int. Conf. on Algebraic Methodology and Software Technology*, pages 258–273, 2004.

[67] R. Jhala and K. McMillan. Array abstractions from proofs. In *Int. Conf. on Computer Aided Verification*, 2007.

[68] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[69] N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2004. http://www.cs.wisc.edu/wpis/wpds++/.

[70] G. Kildall. A unified approach to global program optimization. In *Symp. on Princ. of Prog. Lang.*, pages 194–206, 1973.

[71] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Int. Conf. on Comp. Construct.*, pages 125–140, 1992.

[72] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Security Symp.*, 2005.

[73] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *Int. Conf. on Computer Aided Verification*, pages 135–147, 2004.

[74] A. Lal and T. Reps. Improving pushdown system model checking. In *Int. Conf. on Computer Aided Verification*, pages 343–357, 2006.

[75] A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Int. Conf. on Computer Aided Verification*, pages 434–448, 2005.

[76] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[77] W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Conf. on Prog. Lang. Design and Impl.*, pages 235–248, 1992.

[78] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.

[79] A. Loginov. *Refinement-based program verification via three-valued-logic analysis*. PhD thesis, Comp. Sci. Dept. Univ. of Wisconsin, 2006.

[80] F. Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *Int. Conf. on Supercomputing*, pages 226–235, 1992.

[81] F. Masdupuy. *Array Indices Relational Semantic Analysis using Rational Cosets and Trapezoids*. PhD thesis, Ecole Polytechnique, 1993.

[82] F. Masdupuy. Semantic analysis of interval congruences. In *Int. Conf. on Formal Methods in Prog. and their Appl.*, pages 142–155, 1993.

[83] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symp. on Programming*, pages 5–20, 2005.

[84] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Prog. as Data Objects*, pages 155–172, 2001.

[85] A. Mine. The octagon abstract domain. In *Proc. Eighth Working Conf. on Rev. Eng.*, pages 310–322, 2001.

[86] A. Mine. A few graph-based relational numerical abstract domains. In *Static Analysis Symp.*, pages 117–132, 2002.

[87] A. Mine. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Normale Supérieure, 2004.

[88] E. Myers. A precise interprocedural data flow algorithm. In *Symp. on Princ. of Prog. Lang.*, pages 219–230, 1981.

[89] J. Obdrzalek. Model checking java using pushdown systems. In *Workshop on Formal Techniques for Java-like Programs*, 2002.

[90] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[91] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, 2005.

[92] R.Alur and D. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[93] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Symp. on Princ. of Prog. Lang.*, pages 119–132, 1999.

[94] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang.*, pages 49–61, 1995.

[95] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symp. on Programming*, pages 380–398, 2003.

[96] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation*, pages 252–266, 2004.

[97] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comput. Program.*, 58(1-2):206–263, 2005.

[98] A. Rountev and B.G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Int. Conf. on Comp. Construct.*, pages 20–36, 2001.

[99] A. Rountev, B.G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *Int. Symp. on Foundations of Software Engineering*, pages 235–252, 1999.

[100] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[101] S. Sankaranarayanan, M. Colón, H. Sipma, and Z. Manna. Efficient strongly relational polyhedral analysis. In *Verification, Model Checking, and Abstract Interpretation*, pages 111–125, 2006.

[102] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *Static Analysis Symp.*, pages 3–17, 2006.

[103] S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Verification, Model Checking, and Abstract Interpretation*, pages 25–41, 2005.

[104] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.

[105] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symp. on Princ. of Prog. Lang.*, pages 1–14, 1997.

[106] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[107] A. Simon and A. King. Analyzing string buffers in C. In *Int. Conf. on Algebraic Methodology and Software Technology*, pages 365–379, 2002.

[108] A. Simon and A. King. Widening polyhedra with landmarks. In *Asian Symp. on Prog. Lang. and Syst.*, pages 166–182, 2006.

[109] A. Simon, A. King, and J. Howe. Two variables per linear inequality as an abstract domain. In *Logic-Based Program Synthesis and Tranformation*, pages 71–89, 2002.

[110] P. Černý. Verification by abstract interpretation of parametrized predicates, 2003. Available at “http://www.cis.upenn.edu/ cernyp/”.

[111] P. Černý. Vérification par interprétation abstraite de prédicats paramétriques. D.E.A. Report, Univ. Paris VII & École normale supérieure, September 2003.

[112] A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Static Analysis Symp.*, pages 36–51, 2002.

[113] A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *Static Analysis Symp.*, pages 149–164, 2004.

[114] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symp. on Network and Distributed Systems Security*, February 2000.

[115] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conf. on Prog. Lang. Design and Impl.*, 1995.

[116] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *Symp. on Princ. of Prog. Lang.*, pages 351–363, 2005.

[117] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Static Analysis Symp.*, pages 69–84, 2002.

[118] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 530–545, 2004.

APPENDIX
Proofs of Several Lemmas and Theorems

Lemma 3.6 Let $S^\flat \in \wp(U^\sharp \to \mathbb{V})$ be a partial abstract state. And let $\phi \in \Phi$ be an arbitrary expression. Then,

$$\forall f \in S^\flat \; \left[\textit{Values}_{S^\flat,\phi}(f) = [\![\textit{Values}_{S^\flat,\phi}]\!]^\flat(f)\right].$$

Proof.

Pick an arbitrary function $f \in S^\flat$.

(Soundness). First, let's show that $\textit{Values}_{S^\flat,\phi}(f) \subseteq [\![\textit{Values}_{S^\flat,\phi}]\!]^\flat(f)$. Let's pick a value $a \in \textit{Values}_{S^\flat,\phi}(f)$. From Eqn. (3.3), it follows that there is a concrete state $S \in (\gamma_1 \circ \gamma_2)(S^\flat)$, such that $f \in (\alpha_2 \circ \alpha_1)(S)$ and $a = [\![\phi]\!]_{ND}(S(\sigma_S(w_1)), \ldots, S(\sigma_S(w_k)))$. Recall our assumption that first $\hat{k}$ of w_i are mapped to *summary* objects. Let's construct a function $f' : U^\sharp_\phi \to \mathbb{V}$ as follows:

$$f'(u^\sharp_i) = \begin{cases} f(u^\sharp_i) & \text{if } 1 \leq i \leq m \\ S(\sigma_S(w_{i-m})) & \text{if } m+1 \leq i \leq m+\hat{k} \end{cases}$$

Two things are of interest about f'. First, trivially, $\forall u^\sharp \in U^\sharp \; \left[f'(u^\sharp) = f(u^\sharp)\right]$. Second,

$$[\![\phi]\!]_{ND}(f'(\sigma^\sharp_\phi(w_1)), \ldots, f'(\sigma^\sharp_\phi(w_k))) = a$$

To see this, recall the definition of $\sigma^\sharp_\phi$:

- for $i \in [1, \hat{k}]$, $\sigma^\sharp_\phi(w_i) = u^\sharp_{m+i}$, thus $f'(\sigma^\sharp_\phi(w_i)) = f'(u^\sharp_{m+i}) = S(\sigma_S(w_i))$;
- for $i \in [\hat{k}+1, k]$, $\sigma^\sharp_\phi(w_i) = \sigma^\sharp(w_i) \in \left\{u^\sharp_1, \ldots u^\sharp_m\right\}$; however, abstract object $\sigma^\sharp(w_i)$ is non-summary; thus the following relationship holds:

$$f'(\sigma^\sharp_\phi(w_i)) = f'(\sigma^\sharp(w_i)) = f(\sigma^\sharp(w_i)) = f(\pi_S(\sigma_S(w_i))) = S(\sigma_S(w_i)).$$

> This holds, because the first abstraction step maps each *non-summary* abstract object $u^\sharp$ to the *singleton* set $\{S(u)\}$ (where $\pi_S(u) = u^\sharp$), and the second abstraction step creates a set of functions, each of which maps $u^\sharp$ to $S(u)$.

Thus, $[\![\phi]\!]_{ND}(f'(\sigma^\sharp_\phi(w_1)), \ldots, f'(\sigma^\sharp_\phi(w_k))) = [\![\phi]\!]_{ND}(S(\sigma_S(w_1)), \ldots, S(\sigma_S(w_k))) = a.$

In the next step of the proof, we will show that $f' \in [\![\mathit{expand}_\phi]\!](S^\flat)$. We will show this by induction on the free-variable subscript i (that is, we will consider the sequence of expanded function sets constructed by each consecutive application of the *expand* operation in the definition *expand*$_\phi$, and, on each step i, we will show that there is a function f_i in the resulting set that agrees with f' on objects $u^\sharp_0$ to $u^\sharp_{m+i}$).

Base case: $i = 1$. Let $u^\sharp = \sigma^\sharp(w_1)$. The object $u^\sharp$ is summary. Also, let $u = \sigma_S(w_1)$ be the concrete object to which variable w_1 is bound in the concrete state S (clearly, $\pi_S(u) = u^\sharp$). Consider the set $(\alpha_2 \circ \alpha_1)(S)$. Clearly, $f \in (\alpha_2 \circ \alpha_1)(S)$. The second abstraction step generates *all* possible mappings of abstract objects to the values of the concrete objects they represent. Thus, trivially, there must be a function $g \in (\alpha_2 \circ \alpha_1)(S^\flat)$, such that

$$g(t^\sharp) = \begin{cases} S(u) & \text{if } t^\sharp = u^\sharp \\ f(t^\sharp) & \text{otherwise} \end{cases}$$

That is, in function g, all abstract objects are mapped to the values of the same concrete objects as in function f with the exception of $u^\sharp$, which is mapped to the value of the concrete object u (technically, g could equal f). Since $(\alpha_2 \circ \alpha_1)(S) \subseteq S^\flat$, it follows that $g \in S^\flat$.

Let $S^\flat_1 = [\![\mathit{expand}_{\sigma^\sharp(w_1),u^\sharp_{m+1}}]\!](S^\flat)$. It follows directly from the definition of *expand*, that there is a function $f_1 \in S^\flat_1$ (constructed from functions f and g), such that

$$f_1(u^\sharp_{m+1}) = g(u^\sharp) = S(u) = S(\sigma_S(w_1)) \quad \text{and} \quad \forall t^\sharp \in U^\sharp \ \left[f_1(t^\sharp) = f(t^\sharp)\right]$$

Thus, the function $f_1 \in [\![\mathit{expand}_{\sigma^\sharp(w_1),u^\sharp_{m+1}}]\!](S^\flat)$ agrees with the function f' on abstract objects $u^\sharp_1$ through $u^\sharp_{m+1}$.

Inductive case. Assume that there is a function $f_i \in [\![\mathit{expand}_{\phi,i}]\!](S^\flat)$, where the operation *expand*$_{\phi,i}$ denotes the composition of the first i *expand* operations in the definition of *expand*$_\phi$.

Also, assume that f_i agrees with f' on the abstract objects $u_1^\sharp$ through $u_{m+i}^\sharp$. We need to show that there is a function $f_{i+1} \in [\![\textit{expand}_{\phi,i+1}]\!](S^\flat)$ that agrees with f' on the abstract objects $u_1^\sharp$ through $u_{m+i+1}^\sharp$.

The reasoning in this case is very similar to the reasoning in the base case. Let $u^\sharp = \sigma^\sharp(w_{i+1})$. The object $u^\sharp$ is summary. Also, let $u = \sigma_S(w_{i+1})$ be the concrete object that is assigned to variable w_i in the concrete state S. By the same reasoning as in the base case, there must be a function $g \in (\alpha_2 \circ \alpha_1)(S^\flat)$, such that

$$g(t^\sharp) = \begin{cases} S(u) & \text{if } t^\sharp = u^\sharp \\ f(t^\sharp) & \text{otherwise} \end{cases}$$

Furthermore, due to the symmetries of the expand operation and the abstraction, there must be a function $g_i \in [\![\textit{expand}_{\phi,i}]\!](S^\flat)$, such that for all $t^\sharp \in \{u_1^\sharp, \ldots, u_{m+i}^\sharp\}$, the following holds:

$$g_i(t^\sharp) = \begin{cases} S(u) & \text{if } t^\sharp = u^\sharp \\ f_i(t^\sharp) & \text{otherwise} \end{cases}$$

Intuitively, this function is the result of g's participation in the same "function pairings" (in the definition of *expand*) that constructed the function f_i from the function f. The combination of the functions f_i and g_i by the *expand* operation yields the function f_{i+1}, such that

$$f_{i+1}(u_{m+i+1}^\sharp) = g(u^\sharp) = S(u) = S(\sigma_S(w_i)) \quad \text{and} \quad \forall t^\sharp \in \{u_1^\sharp, \ldots, u_{m+i}^\sharp\} \ \left[f_{i+1}(t^\sharp) = f_i(t^\sharp)\right].$$

Note that f_{i+1} agrees with the function f' on abstract objects $u_1^\sharp$ through $u_{m+i}^\sharp$. Thus, after $\hat{k}$ induction steps, the function $f_{\hat{k}} \in [\![\textit{expand}_\phi]\!](S^\flat)$ is constructed, such that $f' = f_{\hat{k}}$.

We have shown that there exists $f' \in [\![\textit{expand}_\phi]\!](S^\flat)$, such that

$$[\![\phi]\!]_{ND}(f'(\sigma_\phi^\sharp(w_1)), \ldots, f'(\sigma_\phi^\sharp(w_k))) = a \quad \text{and} \quad \forall u^\sharp \in U^\sharp \ \left[f'(u^\sharp) = f(u^\sharp)\right].$$

From the definition of $[\![\textit{Values}_{S^\flat,\phi}]\!]^\flat$, it follows that $a \in [\![\textit{Values}_{S^\flat,\phi}]\!]^\flat$. Thus, we conclude that $\textit{Values}_{S^\flat,\phi}(f) \subseteq [\![\textit{Values}_{S^\flat,\phi}]\!]^\flat(f)$ for any $f \in S^\flat$.

(Completeness). Next, we need to show that $[\![\textit{Values}_{S^\flat,\phi}]\!]^\flat(f) \subseteq \textit{Values}_{S^\flat,\phi}(f)$. Lets pick a value $b \in [\![\textit{Values}_{S^\flat,\phi}]\!]^\flat(f)$. According to the definition of $[\![\textit{Values}_{S^\flat,\phi}]\!]^\flat(f)$, there exists a function $f' \in$

$[\![\textit{expand}_\phi]\!](S^\flat)$, such that

$$[\![\phi]\!]_{ND}(f'(\sigma^\sharp_\phi(w_1)), \ldots, f'(\sigma^\sharp_\phi(w_k))) = b \quad \text{and} \quad \forall u^\sharp \in U^\sharp \ \left[f'(u^\sharp) = f(u^\sharp)\right].$$

We will use f' to manufacture a concrete state $S \in (\gamma_1 \circ \gamma_2)(S^\flat)$, such that

$$[\![\phi]\!]_{ND}(S(\sigma_S(w_1)), \ldots, S(\sigma_S(w_k))) = b \quad \text{and} \quad f \in (\alpha_2 \circ \alpha_1)(S). \tag{A.1}$$

Let the concrete universe of state S be $U_S = \left\{u_1, ..., u_{m+\hat{k}}\right\}$. We will define the state S as follows:

$$S(u_i) = \begin{cases} f(u^\sharp_i) & \text{if } i \in [1, m] \\ f'(\sigma^\sharp_\phi(w_{i-m})) & \text{if } i \in [m+1, m+\hat{k}] \end{cases}$$

Let π_S be defined as follows:

$$\pi_S(u_i) = \begin{cases} u^\sharp_i & \text{if } i \in [1, m] \\ \sigma^\sharp(w_{i-m}) & \text{if } i \in [m+1, m+\hat{k}] \end{cases}$$

Finally, let σ_S be defined as follows

$$\sigma_S(w_i) = \begin{cases} u_{m+i} & \text{if } i \in [1, \hat{k}] \\ \sigma^\sharp(w_i) & \text{if } i \in [\hat{k}+1, k] \end{cases}$$

Note that the two conditions in Eqn. (A.1) hold by construction. Showing that $S \in (\gamma_1 \circ \gamma_2)(S^\flat)$ is more complicated. We will show this, by showing that $(\alpha_2 \circ \alpha_1)(S) \subseteq S^\flat$.

Pick a function $g \in (\alpha_2 \circ \alpha_1)(S)$. Note that functions g and f agree on objects $u^\sharp \in U^\sharp$, such that $u^\sharp \neq \sigma^\sharp(w_i)$ for all $i \in [1, \hat{k}]$. This follows from the abstraction: all such objects $u^\sharp$ are non-summary with respect to S and π_S, and the corresponding objects $u \in U_S$ that they represent are mapped to $f(u^\sharp)$ (by the construction of S). For objects $u^\sharp \in U^\sharp$, such that $u^\sharp = \sigma^\sharp(w_i)$ for some $i \in [1, \hat{k}]$, $g(u^\sharp)$ is equal to either $f(u^\sharp)$ or some value in the set $\{f'(u^\sharp_{m+j}) \mid \sigma^\sharp(w_j) = u^\sharp\})$. Note, also, that all such $u^\sharp$ are summary objects with respect to $S^\flat$ (because of the assumption that the first $\hat{k}$ of variables w_i are mapped to summary objects). Thus, the functions f and g may disagree on at most $\hat{k}$ objects in $U^\sharp$.

We proceed as follows: for each number of disagreements r between the functions f and g, starting with zero, we show that there must be a function $g_r \in S^\flat$, such that $g = g_r$. Below, we will detail the first three cases, other cases up to $r = \hat{k}$ use the same reasoning. After the case $r = \hat{k}$ is considered, we will have covered all possibilities for function g, and thus, g must be in $S^\flat$.

Case $r = 0$**.** This case is trivial: the functions g and f agree on all objects in $U^\sharp$. Thus, $g = f$, and consequently $g \in S^\flat$.

Case $r = 1$**.** The functions f and g disagree on the mapping of one object. We will denote that object $u^\sharp$. Without loss of generality, let us assume that g maps the object $u^\sharp$ to the value of the concrete object $u_{m+i} \in U_S$, such that $\pi_S(u_{m+i}) = u^\sharp$. That is, $g(u^\sharp) = S(u_{m+i})$. The following facts follow from the construction of the state S:

$$\text{(i)} \quad S(u_{m+i}) = f'(\sigma^\sharp_\phi(w_i)) = f'(u^\sharp_{m+i}) \quad \text{and} \quad \sigma^\sharp(w_i) = u^\sharp.$$

The first equality follows directly from the definition of S and from the definition of $\sigma^\sharp_\phi$; the second equality follows from the definition of π_S.

Next, for $f'(u^\sharp_{m+i})$ to equal $S(u_{m+i})$, it must have been put there by the application of the i-th *expand* operation in the definition of $[\![\textit{expand}_\phi]\!]$. That is, there must be two functions f_{i-1}, g_{i-1} in the set $S^\flat_{i-1} = [\![\textit{expand}_{\phi,i-1}]\!](S^\flat)$, such that f_{i-1} agrees with f' on all objects in the set $\{u^\sharp_1, \ldots, u^\sharp_{m+i-1}\}$, and the function g_{i-1} agrees with the function f_{i-1} on all objects, except for the object $\sigma^\sharp(w_i) = u^\sharp$. Moreover, $g_{i-1}(u^\sharp)$ equals $S(u_{m+i})$. But the function g_{i-1} could have only been produced (by the *expand* sequence) from the function $g_1 \in S^\flat$, such that g_1 and g_{i-1} agree on all objects in $U^\sharp$. Thus, it follows that, for all $t^\sharp \in U^\sharp$,

$$g_1(t^\sharp) = \begin{cases} g_1(u^\sharp) = S(u_{m+i}) = g(u^\sharp) & \text{if } t^\sharp = u^\sharp \\ g_1(t^\sharp) = g_{i-1}(t^\sharp) = f_{i-1}(t^\sharp) = f'(t^\sharp) = f(t^\sharp) = g(t^\sharp) & \text{otherwise} \end{cases}$$

Thus, $g = g_1 \in S^\flat$.

Case $r = 2$**.** The functions f and g disagree on the mapping of two objects, $u^\sharp$ and $v^\sharp$. Without loss of generality, let us assume that g maps the objects $u^\sharp$ and $v^\sharp$ to the values of concrete objects u_{m+i} and u_{m+j} in U_S, respectively; furthermore, let $i < j$. Similarly to the previous case we have:

$$\begin{array}{ll} \text{(i)} & S(u_{m+i}) = f'(\sigma^\sharp_\phi(w_i)) = f'(u^\sharp_{m+i}) \quad \text{and} \quad \sigma^\sharp(w_i) = u^\sharp; \\ \text{(ii)} & S(u_{m+j}) = f'(\sigma^\sharp_\phi(w_j)) = f'(u^\sharp_{m+j}) \quad \text{and} \quad \sigma^\sharp(w_j) = v^\sharp. \end{array}$$

For $f'(u^\sharp_{m+j})$ to equal $S(u_{m+j})$, it must have been put there by the application of the j-th *expand* operation in the definition of $\llbracket \mathit{expand}_\phi \rrbracket$. That is, there must be two functions f_{j-1}, g_{j-1} in the set $S^\flat_{j-1} = \llbracket \mathit{expand}_{\phi,j-1} \rrbracket(S^\flat)$, such that f_{j-1} agrees with f' on all objects in the set $\{u^\sharp_1, \ldots, u^\sharp_{m+j-1}\}$, and the function g_{j-1} agrees with the function f_{j-1} on all objects, except for the object $\sigma^\sharp(w_j) = v^\sharp$. Moreover, $g_{j-1}(v^\sharp)$ equals $S(u_{m+j})$.

Next, recall our assumption that $i < j$. Thus, $f_{j-1}(u^\sharp_{m+i}) = g_{j-1}(u^\sharp_{m+i}) = S(u_{m+i})$. For $g_{j-1}(u^\sharp_{m+i})$ to be equal to $S(u_{m+i})$, it must have been put there by the application of the i-th *expand* operation in the definition of $\llbracket \mathit{expand}_\phi \rrbracket$. That is, there must be two functions g_{i-1}, h_{i-1} in the set $S^\flat_{i-1} = \llbracket \mathit{expand}_{\phi,i-1} \rrbracket(S^\flat)$, such that g_{i-1} agrees with g_{j-1} on all objects in the set $\{u^\sharp_1, \ldots, u^\sharp_{m+i-1}\}$ (most importantly, both functions map object $v^\sharp$ to $S(u_{m+j})$), and the function h_{i-1} agrees with the function g_{i-1} on all objects, except for the object $\sigma^\sharp(w_i) = u^\sharp$: that is, $h_{i-1}(u^\sharp)$ equals $S(u_{m+i})$, and $h_{i-1}(v^\sharp)$ equals $S(u_{m+j})$.

But the function h_{i-1} could have only been produced (by the *expand* sequence) from the function $h_2 \in S^\flat$, such that h_2 and h_{i-1} agree on all objects in $U^\sharp$. Thus, it follows that, for all $t^\sharp \in U^\sharp$,

$$h_2(t^\sharp) = \begin{cases} h_2(u^\sharp) = S(u_{m+i}) = g(u^\sharp) & \text{if } t^\sharp = u^\sharp \\ h_2(v^\sharp) = S(u_{m+j}) = g(v^\sharp) & \text{if } t^\sharp = u^\sharp \\ h_2(t^\sharp) = h_{i-1}(t^\sharp) = g_{i-1}(t^\sharp) = g_{j-1}(t^\sharp) = f_{j-1}(t^\sharp) = f'(t^\sharp) = f(t^\sharp) = g(t^\sharp) & \text{otherwise} \end{cases}$$

Thus, $g = h_2 \in S^\flat$.

Cases $r = 3..\hat{k}$. The remaining cases follow the same reasoning at the first three cases, Note that the number of functions that must be considered to show that function g is in the set $S^\flat$ increases with every step. After the case $r = \hat{k}$ is considered, all the possibilities for the function g are covered. Thus, all functions $g \in (\alpha_2 \circ \alpha_1)(S)$ are also in $S^\flat$, and therefore, $S \in (\gamma_1 \circ \gamma_2)(S^\flat)$.

We have shown that $S \in (\gamma_1 \circ \gamma_2)(S^\flat)$. The two conditions in Eqn. (A.1) hold by construction. Thus, by the definition of $\mathit{Values}_{S^\flat,\phi}$, we can conclude that $b \in \mathit{Values}_{S^\flat,\phi}(f)$. Therefore, $\llbracket \mathit{Values}_{S^\flat,\phi} \rrbracket^\flat(f) \subseteq \mathit{Values}_{S^\flat,\phi}(f)$.

■

Theorem 3.7 The abstract transformer $[\![x \leftarrow \phi]\!]^\flat$ is *sound*. That is, for an arbitrary partial abstract state $S^\flat$,

$$(\gamma_1 \circ \gamma_2)([\![x \leftarrow \phi]\!]^\flat(S^\flat)) \supseteq [\![x \leftarrow \phi]\!]((\gamma_1 \circ \gamma_2)(S^\flat)).$$

Proof. Let $S^\flat$ be a partial abstract state, and let $S_1^\flat$ be the result of applying the abstract transformer for the assignment transition to $S^\flat$, that is $S_1^\flat = [\![x \leftarrow \phi]\!]^\flat(S^\flat)$. Let S denote an arbitrary concrete state represented by $S^\flat$, that is, $S \in (\gamma_1 \circ \gamma_2)(S^\flat)$. Let S_1 denote the concrete state obtained by applying the assignment transition to the state S, i.e., $S_1 = [\![x \leftarrow \phi]\!](S)$. We need to show that $S_1 \in (\gamma_1 \circ \gamma_2)(S_1^\flat)$.

Let $[\![\phi]\!]_{ND}(S(\sigma_S(w_1)), \ldots, S(\sigma_S(w_k))) = a$, where a is some value in $\mathbb{V}$. The state S_1 has the same universe as S, that is $U_{S_1} = U_S$. Furthermore, for all $u \in U_S$, $S_1(u)$ is equal to $S(u)$, except for the object $v = \sigma_S(x)$ (i.e., the object that is updated by the assignment): $S_1(v) = a$.

Recall that each function in the partial abstraction of a concrete state S is obtained by mapping each abstract object $u^\sharp$ to the value of a non-deterministically chosen concrete object in $\pi_S^{-1}(u^\sharp)$. Let us consider functions f and f_1 in the abstractions of S and S_1, respectively, that are constructed by choosing the same mapping in S and S_1 for each abstract object in $U^\sharp$. Obviously, f and f_1 map all objects in $U^\sharp$ to the same values, except for the object $\pi_S(v) = \sigma^\sharp(x)$, which is mapped to $S(v)$ by f, and to a by f_1.

Clearly, $f \in S^\flat$ because $S \in (\gamma_1 \circ \gamma_2)(S^\flat)$. Also, $a \in \mathit{Values}_{S^\flat,\phi}(f)$ by Eqn. (3.3). Thus, by the definition of $[\![x \leftarrow \phi]\!]^\flat$ in Eqn. (3.5), there must be a function $f' \in S_1^\flat$, such that $f'(\sigma^\sharp(x)) = a$ and for all other objects $u^\sharp \in U^\sharp$, $f'(u^\sharp) = f(u^\sharp)$. But $f' = f_1$. Thus $f_1 \in S_1^\flat$. Therefore, $(\alpha_2 \circ \alpha_1)(S_1) \subseteq S_1^\flat$, and consequently, $S_1 \in (\gamma_1 \circ \gamma_2)(S_1^\flat)$, which concludes the proof.

■

Lemma 3.8 Let $S^\flat \in \wp(U^\sharp \to \mathbb{V})$ be a partial abstract state. And let $\psi \in \Psi$ be an arbitrary expression. Then,

$$\forall f \in S^\flat \; \left[\mathit{Values}_{S^\flat,\psi}(f) = [\![\mathit{Values}_{S^\flat,\psi}]\!]^\flat(f)\right].$$

Proof.

The proof follows exactly the same argument as the proof for Lem. 3.6, with the only exception that the values a and b selected from $\mathit{Values}_{S^\flat,\psi}(f)$ and $[\![\mathit{Values}_{S^\flat,\psi}]\!]^\flat(f)$, respectively, are Boolean values, rather than values from $\mathbb{V}$.

■

Theorem 3.9 The abstract transformer $[\![\mathit{assume}(\psi)]\!]^\flat$ is *sound*. That is, for an arbitrary partial abstract state $S^\flat$,

$$(\gamma_1 \circ \gamma_2)([\![\mathit{assume}(\psi)]\!]^\flat(S^\flat)) \supseteq [\![\mathit{assume}(\psi)]\!]((\gamma_1 \circ \gamma_2)(S^\flat)).$$

Proof. Let $S^\flat$ be a partial abstract state, and let $S_1^\flat$ be the result of applying the abstract transformer for the assume transition to $S^\flat$, that is, $S_1^\flat = [\![\mathit{assume}(\psi)]\!]^\flat(S^\flat)$. Let S denote an arbitrary concrete state represented by $S^\flat$ (that is, $S \in (\gamma_1 \circ \gamma_2)(S^\flat)$), such that S satisfies the condition ψ, i.e., $[\![\mathit{assume}(\psi)]\!](S) = \mathit{true}$. We need to show that $S \in (\gamma_1 \circ \gamma_2)(S_1^\flat)$.

Consider an arbitrary function f in the abstraction of S: that is, $f \in (\alpha_2 \circ \alpha_1)(S)$. Because S satisfies the condition ψ (i.e., $[\![\mathit{assume}(\psi)]\!](S) = \mathit{true}$), the following holds by the definition in Eqn. (3.6): $\mathit{true} \in \mathit{Values}_{S^\flat,\psi}(f)$. Thus, by the definition of the abstract transformer in Eqn. (3.7), $f \in S_1^\flat$. Therefore, we can conclude that $(\alpha_2 \circ \alpha_1)(S) \subseteq S_1^\flat$, and consequently, that $S \in (\gamma_1 \circ \gamma_2)(S_1^\flat)$. This concludes the proof of soundness.

■